

HOPL

History of Programming Languages Conference



June 9-10, 2007
San Diego, CA



Sponsored by
ACM SIGPLAN
in cooperation with
SIGSOFT

www.acm.org/sigplan/hopl

FCRC '07

FCRC '07

Proceedings

The Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III)

San Diego, California, USA
9-10 June 2007



in cooperation with ACM SIGSOFT

(co-located with FCRC 2007, 9-16 June 2007)

Copyright © 2007 Association for Computing Machinery, Inc. (ACM)

History of Programming Languages Conference: HOPL-III Co-Chairs Introduction

In 1978, at the first HOPL conference, Jean Sammet wrote:

I'm happy to have this opportunity to open this Conference on the History of Programming Languages. It's something that I personally have been thinking about for almost seven years although the actual work on this has only been going on for the past one-and-a-half years. Most of you have not had the opportunity to look at the Preprints, and perhaps you haven't had too much time to even look at the program. For that reason I want to make sure that you understand something about what *is* intended, and frankly what is *not* intended be done in this conference. We view this as a start: perhaps the first of a number of conferences which will be held about various aspects of the computing field in general, software in particular, and even more specifically, programming languages. We hope that this conference and the Preprints and the final proceedings will stimulate many older people, many younger people, and many of those in the in-between category in the field to do work in history.

This conference, I repeat again, is certainly not the last set of words to be held on this subject. It's only a beginning. I want to emphasize also that it's *not* a conference on the entire history of programming languages, nor even on the entire history of the languages that we selected. As many of you have seen from some of the earlier publicity that we put out, we're trying to consider and report on the technical factors which influenced the development of languages which satisfied a number of criteria. First, they were created and in use by 1967; they remain in use in 1977, which is when we made the decisions; and they've had considerable influence on the field of computing. The criteria for choosing a language include the following factors, although not every factor applied to each language: we considered usage, influence on language design, overall impact on the environment, novelty, and uniqueness. Particularly because of the cut-off date of 1967, some languages, which are in common use today, are not included. We definitely wanted a perspective of 10 years before we started worrying about the early history of the language.

HOPL-I was a start and it did stimulate (some older and younger) people to continue the work of documenting the history of computing in general, and programming languages, in particular. HOPL-II followed in 1993. It extended the notion of history from HOPL-I to include the evolution of languages, language paradigms, and language constructs. It preserved the 10-year perspective. It also repeated the HOPL-I multi-year preparation of submissions, reviews, and re-reviews with teams of reviewers and experts, to come up with the best possible history papers from the people who were directly involved with the creation of their languages.

Fifteen years later, HOPL-III is another step in the documentation of the history of

our field. Work began three years ago in 2004 to create a Program Committee, to establish paper solicitation criteria (see appendix to this proceedings), and to encourage submissions. As with its predecessors, the goal of HOPL-III was to produce an accurate historical record of programming language design and development. To achieve this goal, the Program Committee worked closely with prospective authors and outside experts to help ensure that all the papers were of high quality. As with HOPL-I and II, there were multiple rounds of reviewing to ensure that all the selected papers met requirements for both technical accuracy and historical completeness.

The criteria for the programming languages considered appropriate for HOPL-III were:

1. The programming language came into existence before 1996, that is, it was designed and described at least 11 years before HOPL-III (2007).
2. The programming language has been widely used since 1998 either (i) commercially or (ii) within a specific domain. In either case, “widely used” implies use beyond its creators.
3. There also are some research languages which had great influence on widely used languages that followed them. As long as the research language was used by more than its own inventors, these will be considered to be appropriate for discussion at HOPL-III.

The twelve papers in this proceedings represent original historical perspectives on programming languages that span at least five different programming paradigms and communities: object-oriented, functional, reactive, parallel, and scripting. At the time of the conference, the programming languages community continues to create broader mini-histories of each of those paradigms at <http://en.wikipedia.org/wiki/HOPL>

A conference of this scope and level of preparation could not have happened without the time and assistance of many, many people. First we must thank our colleagues on the program committee

Fran Allen, IBM Research (Emerita)
Thomas J. (Tim) Bergin, American University (Emeritus)
Andrew Black, Portland State University
Koen Claessen, Chalmers University of Technology
Kathleen Fisher, AT&T Research
Susan L. Graham, University of California, Berkeley
Julia Lawall, DIKU
Doug Lea, SUNY Oswego
Peter Lee, Carnegie Mellon University
Michael S. Mahoney, Princeton University

Guy Steele, Sun Microsystems
Benjamin Zorn, Microsoft Research

and the authors of all the submitted papers.

We must also thank the language experts who helped with the extensive paper reviews: Chris Espinoza, Gilad Bracha, Herb Sutter, Andrew Watson, Ulf Wiger, Vivek Sarkar, Norman Ramsey, Greg Nelson, Craig Chambers, and Kathy Yellick. We also thank the set of experts who helped seed the Wikipedia discussions of the language paradigms: Vijay Saraswat, Bard Bloom, Dipayan Gangopadhyay, and Guido van Rossum. Finally, we would like to thank the staff at ACM Headquarters, the SIGPLAN Executive Committee, the SIGSOFT Executive Committee, and Diana Priore, all of whom made this complex conference possible, and Joshua Hailpern, who designed both the HOPL CACM advertisement and the Proceedings and Final Program cover art.

We also wish to acknowledge the generous financial support for HOPL-III that has been provided by:

- An anonymous donor for multimedia capture/post-processing
- Microsoft Research for manuscript copy-editing and proceedings preparation
- IBM Research for subsidizing student registration and in support of program committee operations, the final program, and the HOPL-III website
- ACM SIGPLAN Executive Committee



This has been an ambitious and lengthy project for us; we are glad to see it successfully completed. We hope you enjoy both the conference presentations and the papers in these proceedings – a (partial) record of the past 15 years of our programming languages community.

Barbara Ryder, Rutgers University
Brent Hailpern, IBM Research
HOPL-III Conference/Program Committee co-Chairs

HOPL-III Agenda: Saturday, 9 June 2007

08:45 - 09:00 Introduction

09:00 - 10:00 Keynote by Guy Steele (Sun Microsystems) and Richard P. Gabriel (IBM Research)

10:00 - 10:30 Break

10:30 - 12:20 "**AppleScript**" by William R. Cook (University of Texas at Austin)

"**The evolution of Lua**" by Roberto Ierusalimschy (PUC-Rio), Luiz Henrique de Figueiredo (IMPA), and Waldemar Celes (PUC-Rio)

12:20 - 13:30 Lunch

13:30 - 15:20 "**A history of Modula-2 and Oberon**" by Niklaus Wirth (ETH Zurich)

"**Evolving a language in and for the real world: C++ 1991–2006**" by Bjarne Stroustrup (Texas A&M University and AT&T Labs - Research)

15:20 - 16:00 Break

16:00 - 17:50 "**Statecharts in the making: a personal account**" by David Harel (Weizmann Institute of Science)

"**A history of Erlang**" by Joe Armstrong (Ericsson AB)

HOPL-III Agenda: Sunday, 10 June 2007

08:00 - 08:15 Introduction

08:15 - 10:05 **"The rise and fall of High Performance Fortran: an historical object lesson"** by Ken Kennedy (Rice University), Charles Koelbel (Rice University), Hans Zima (Institute of Scientific Computing, University of Vienna and Jet Propulsion Laboratory, California Institute of Technology)

"The design and development of ZPL" by Lawrence Snyder (University of Washington)

10:05 - 10:30 Break

10:30 - 12:20 **"Self"** by David Ungar (IBM Research), Randall B. Smith (Sun Microsystems)

"The when, why and why not of the BETA programming language" by Bent Bruun Kristensen (University of Southern Denmark), Ole Lehrmann Madsen (University of Aarhus), Birger Møller-Pedersen (University of Oslo)

12:20 - 13:30 Lunch

13:30 - 15:20 **"The development of the Emerald programming language"** by Andrew P. Black (Portland State University), Norman C. Hutchinson (University of British Columbia), Eric Jul (University of Copenhagen) and Henry M. Levy (University of Washington)

"A History of Haskell: being lazy with class" by Paul Hudak (Yale University), John Hughes (Chalmers University), Simon Peyton Jones (Microsoft Research), and Philip Wadler (University of Edinburgh)

15:20 - 15:40 Break

15:40 - 17:00 **Panel: Programming Language Paradigms: Past, Present, and Future**

Kathleen Fisher (AT&T Labs - Research), chair

Bertrand Meyer (ETH Zurich)

Olin Shivers (Georgia Institute of Technology)

Larry Wall

Kathy Yelick (University of California, Berkeley)

AppleScript

William R. Cook

University of Texas at Austin

wcook@cs.utexas.edu

With contributions from Warren Harris, Kurt Piersol, Dave Curbow, Donn Denman, Edmund Lai, Ron Lichty, Larry Tesler, Donald Olson, Mitchell Gass and Eric House

Abstract

AppleScript is a scripting language and environment for the Mac OS. Originally conceived in 1989, AppleScript allows end users to *automate* complex tasks and *customize* Mac OS applications. To automate tasks, AppleScript provides standard programming language features (control flow, variables, data structures) and sends Apple Events to invoke application behavior. Apple Events are a variation on standard remote procedure calls in which messages can identify their arguments by queries that are interpreted by the remote application. This approach avoids the need for remote object pointers or proxies, and reduces the number of communication round trips, which are expensive in high latency environments like the early Macintosh OS. To customize an application that uses AppleScript's Open Scripting Architecture, users attach scripts to application objects; these scripts can then intercept and modify application behavior.

AppleScript was designed for casual users: AppleScript syntax resembles natural language, and scripts can be created easily by recording manual operations on a graphical interface. AppleScript also supported internationalization in allowing script to be presented in multiple dialects, including English, Japanese, or French. Although the naturalistic syntax is easy to read, it can make scripts much more difficult to write.

Early adoption was hindered by the difficulty of modifying applications to support Apple Events and the Open Scripting Architecture. Yet AppleScript is now widely used and is an essential differentiator of the Mac OS. AppleScript's communication model is a precursor to web services, and the idea of embedded scripting has been widely adopted.

Permission to make digital/hard copy of part of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date of appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Permission may be requested from the Publications Dept., ACM, Inc., 2 Penn Plaza, New York, NY 11201-0701, USA, fax:+1(212) 869-0481, permissions@acm.org.

©2007 ACM 978-1-59593-766-7/2007/06-ART1 \$5.00

DOI 10.1145/1238844.1238845

<http://doi.acm.org/10.1145/1238844.1238845>

Categories and Subject Descriptors D.3 [Programming Languages]

General Terms Languages, Design, Human Factors

Keywords AppleScript, Scripting, History

1. Introduction

The development of AppleScript was a long and complex process that spanned multiple teams, experienced several false starts and changes of plan, and required coordination between different projects and companies. It is difficult for any one person, or even a small group of people, to present a comprehensive history of such a project, especially without official support from the company for which the work was done. The email record of the team's communications have been lost, and the author no longer has access to internal specifications and product plans.

Nevertheless, I believe that the development of AppleScript is a story worth telling, and I have been encouraged to attempt it despite the inherent difficulty of the task. I can offer only my own subjective views on the project, as someone who was intimately involved with all its aspects. I apologize in advance for errors and distortions that I will inevitably introduce into the story, in spite of my best efforts to be accurate.

I first heard the idea of AppleScript over lunch with Kurt Piersol in February of 1991. The meeting was arranged by our mutual friend James Redfern. I knew James from Brown, where he was finishing his undergraduate degree after some time off, and I was working on my PhD. James and I both moved to California at about the same time, in 1988. We spent a lot of time together and I had heard a little about what he was up to, but he claimed it was secret. James arranged the meeting because Kurt was looking for someone to lead the AppleScript effort, and I was looking for something new to do.

For the previous two and a half years I had been working at HP Labs. I was a member of the Abel group, which included Walt Hill, Warren Harris, Peter Canning, and Walter Olthoff. John Mitchell consulted with us from Stanford. The group was managed by Alan Snyder, whose formaliza-

tion of object concepts [35] was one basis for the development of CORBA. At HP Labs I finished writing my PhD thesis, A Denotational Semantics of Inheritance [14, 19], which the Abel group used as the foundation for a number of papers. We published papers on inheritance and subtyping [18], object-oriented abstraction [7, 16], mixins [5], F-Bounded polymorphism [6], and a fundamental flaw in the Eiffel type system [15]. I continued some of the work, analyzing the Smalltalk collection hierarchy [17], after I left HP.

Near the end of 1990 HP Labs was undergoing a reorganization and I was not sure I would fit into the new plan. In addition, I was interested in making a change. I had developed several fairly large systems as an undergraduate, including a text editor and a graphical software design tool [39, 20] that were used by hundreds of other students for many years. As a graduate student I had focused on theory, but I wanted to learn the process of commercial software development. Apple seemed like a reasonable place to try, so I agreed to talk with Kurt.

1.1 AppleScript Vision—Over Lunch

Kurt and I hit it off immediately at lunch. Kurt Piersol is a large, friendly, eloquent man who was looking for people to work on the AppleScript project. Kurt graduated with a B.S. from the University of Louisville’s Speed Scientific School. He then worked at Xerox, building systems in Smalltalk-80 and productizing research systems from Xerox PARC. Kurt was hired to work on AppleScript in 1989. He was originally asked to work on a development environment for the new language, but eventually took on the role of steering the project.

Kurt and I discussed the advantages or disadvantages of command-line versus graphical user interfaces. With command-line interfaces, commonly used on Unix, users frequently write scripts that automate repeated sequences of program executions. The ability to pipe the output of one program into another program is a simple but powerful form of inter-application communication that allows small programs to be integrated to perform larger tasks. For example, one can write a script that sends a customized version of a text file to a list of users. The sed stream editor can create the customized text file, which is then piped into the mail command for delivery. This new script can be saved as a mail-merge command, so that it is available for manual execution or invocation from other scripts. One appealing aspect of this model is its compositionality: users can create new commands that are invoked in the same way as built-in commands. This approach works well when atomic commands all operate on a common data structure, in this case text streams. It was not obvious that it would work for more complex structured data, like images, databases, or office documents, or for long-running programs that interact with users.

With a graphical user interface (GUI) important functions, including the mail-merge command described above,

are usually built into a larger product, e.g. a word processor. A GUI application offers pre-packaged integrated functionality, so users need not combine basic commands to perform common tasks. Although careful design of graphical interfaces eliminates the need for automation for basic use, there are still many tasks that users perform repeatedly within a graphical interface. The designers of the graphical interface cannot include commands to cover all these situations — if they did, then some users would execute these new commands repeatedly. No finite set of commands can ever satisfy all situations.

Most users are happy with GUI applications and do not need a command-line interface or a scripting language. But there are clearly some limitations to the GUI approach on which Macintosh OS was based. Power users and system integrators were frustrated by the inability to build custom solutions by assembling multiple applications and specializing them to particular tasks. Allowing users to automate the tasks that are relevant to them relieves pressure on the graphical user interface to include more and more specialized features.

The vision for AppleScript was to provide a kind of command-line interface to augment the power of GUI applications and to bring this power to casual users.

1.2 Automation and Customization

Kurt and I talked about two ways in which scripts and GUI applications could interact: for automation and for customization. *Automation* means that a script directs an application to perform a sequence of actions—the actions are performed “automatically” rather than manually. With automation, the script is in control and one or more applications respond to script requests. *Customization* occurs when a script is invoked from within an application—the script can perform “custom” actions that replace or augment the normal application behavior. With customization, the application manages and invokes scripts that users have attached to application objects. Automation is useful even without customization, but customization requires automation to be useful.

We discussed whether there was sufficient benefit in providing a standard platform for scripting, when custom solutions for each application might be better. Some applications already had their own macro capability or a proprietary scripting language. However, this approach requires users to learn a different scripting language to automate each application. These languages typically include some variation on the basic elements of any programming language, including variables, loops, conditionals, data types, procedures, and exceptions. In addition, they may include special forms or constructs specific to the application in question. For example, a spreadsheet language can refer to cells, sheets, equations and evaluation.

One benefit of a standard scripting platform is that applications can then be *integrated* with each other. This capa-

bility is important because users typically work with multiple applications at the same time. In 1990, the user's options for integrating applications on the Macintosh OS were limited to shared files or copy/paste with the clipboard. If a repeated task involves multiple applications, there is little hope that one application will implement a single command to perform the action. Integrating graphical applications can be done at several levels: visual integration, behavioral integration, or data integration. *Visual integration* involves embedding one graphical component inside another; examples include a running Java applet inside a web page, or a specialized organization chart component displayed inside a word-processing document. *Behavioral integration* occurs when two components communicate with each other; examples include workflow or invocation of a spell-check component from within a word processor. *Data integration* occurs whenever one application reads a file (or clipboard data) written by another application. A given system can include aspects of all three forms of integration.

I agreed with Kurt that the most important need at that time was for behavioral integration. To compete with custom application-specific scripting languages, AppleScript would have to allow application-specific behaviors to be incorporated into the language in as natural a way as possible, while preserving the benefits of a common underlying language. The core of the language should support the standard features of a programming language, including variables, procedures, and basic data types. An application then provides a vocabulary of specific terminology that apply to the domain: a photo-processing application would manipulate images, pixels and colors, while a spreadsheet application would manipulate cells, sheets, and graphs. The idea of AppleScript was to implement the “computer science boilerplate” once, while seamlessly integrating the vocabulary of the application domain so that users of the language can manipulate domain objects naturally. We discussed the vision of AppleScript as a pervasive architecture for inter-application communication, so that it is easy to integrate multiple applications with a script, or invoke the functionality of one application from a script in another application. We hoped that scripting would create a “network effect”, by which each new scriptable application improves the value of scripting for all other applications.

1.3 AppleScript Begins

Soon after, Kurt offered me a job and I accepted quickly. This event illustrates one of the recurring characteristics of AppleScript: the basic idea is so compelling that it is enthusiastically embraced by almost every software developer who is exposed to it.

What was not immediately obvious was how difficult the vision was to achieve—not for strictly technical reasons, but because AppleScript required a fundamental refactoring, or at least augmentation, of almost the entire Macintosh code base. The demonstrable benefits of AppleScript’s vision has

led developers to persevere in this massive task for the last twenty years; yet the work is truly Sisyphean, in that the slow incremental progress has been interrupted by major steps backward, first when the hardware was changed from the Motorola 68000 chip to the IBM PowerPC, and again when the operating system was reimplemented for Mac OS X.

At this point it is impossible to identify one individual as the originator of the AppleScript vision. The basic idea is simple and has probably been independently discovered many times. The AppleScript team was successful in elaborating the original vision into a practical system used by millions of people around the world.

2. Background

When I started working at Apple in April 1991 I had never used a Macintosh computer. My first task was to understand the background and context in which AppleScript had to be built.

The main influences I studied were the Macintosh operating system, HyperCard, and Apple Events. HyperCard was a good source of inspiration because it was a flexible application development environment with a scripting language embedded within it. A previous team had designed and implemented Apple Events to serve as the underlying mechanism for inter-application communication. The Apple Events Manager had to be shipped early so that it could be included in the Macintosh System 7 OS planned for summer 1991. When I started at Apple, the Apple Event Manager was in final beta testing. The fact that AppleScript and Apple Events were not designed together proved to be a continuing source of difficulties.

Macintosh systems at that time had 4 to 8 megabytes of random-access memory (RAM) and a 40- to 60-megabyte hard drive. They had 25-50 MHz Motorola 68000 series processors. The entire company was internally testing System 7.0, a major revision of the Macintosh OS.

Applications on the Macintosh OS were designed around a main event processing loop, which handled lower-level keyboard and mouse events from the operating system [12]. The OS allowed an application to post a low-level event to another application, providing a simple form of inter-application communication. In this way one application could drive another application, by sending synthetic mouse and keyboard events that select menus or data and enter text into an application. This technique was used in two utility applications, MacroMaker and QuicKeys, which recorded and played back low-level user interface events. It was also used in the Macintosh help system, which could post low-level events to show the user how to use the system. Scripts that send low-level events are fragile, because they can fail if the position or size of user interface elements changes between the time the script is recorded and when it is run. They are also limited in the actions they can perform; low-

level events can be used to change the format of the current cell in a spreadsheet, but cannot read the contents of a cell.

In the following section I describe these systems as they were described to me at the start of the AppleScript project, in April 1991.

2.1 HyperCard

HyperCard [27, 30], originally released in 1987, was the most direct influence on AppleScript. HyperCard is a combination of a simple database, a collection of user interface widgets, and an English-like scripting language. These elements are organized around the metaphor of information on a collection of index cards. A collection of such cards is called a *stack*. A card could contain text, pictures, buttons, and other graphical objects. Each object has many *properties*, including location on the card, size, font, style, etc. Cards with similar structure could use a common *background*; a background defines the structure, but not the content, of multiple cards. For example, a stack for household information might contain recipe cards and contact cards. The recipe cards use a recipe background that includes text fields for the recipe title, ingredients, and steps. The contact cards use a contact background with appropriate fields, including a photo.

HyperCard scripts are written in HyperTalk, an English-like scripting language [28]. The language is for the most part a standard structured, imperative programming language. However, it introduced a unique approach to data structures: the stack, cards, objects and properties are used to store data. These structures are organized in a containment hierarchy: stacks contain cards, cards contain objects, and properties exist on stacks, cards, and objects. This predefined structure makes it easy to build simple stacks, but more difficult to create custom data structures.

Scripts in a stack can refer to the objects, collections of objects, and their properties by using *chunk expressions*. A chunk expression is best understood as a kind of query. For example, the following chunk expression refers to the text style property of a word element in a field of the current card:

```
the textStyle of word 2  
of card field "Description"
```

A chunk expression can refer to properties and elements of objects. A property is a single-valued attribute, for example `textStyle`. Elements are collections of objects identified by a type, for example `word` and `card field`. Element access may be followed by a name, index or range to select element(s) from the collection. Properties access distributes over collections; the following expression represents a collection of 10 text style properties:

```
the textStyle of character 1 to 10  
of card field "Description"
```

HyperCard has a built-in set of property and collection names.

Each object has a *script* containing procedures defined for that object. If the procedure name is an *event* name, then the procedure is a *handler* for that event—it is called when the event occurs for that object. For example, a button script may have handlers for `mouseDown`, `mouseUp` and `mouseMove` events. The following handler shows the next card when a button is released.

```
on mouseUp  
    go to next card  
end mouseUp
```

Actions can be performed on chunk expressions to modify the stack, its cards, the objects, or their properties. For example, clicking a button may run a script that moves to the next card, adds/removes cards, or modifies the contents of one or more cards. HyperCard has a set of predefined actions, including `set`, `go`, `add`, `close`, etc. For example, the text can be updated to a predefined constant `bold`:

```
set the textStyle of character 1 to 10  
of card field "Description" to bold
```

HyperCard 2.0 was released in 1990. HyperCard was very influential and widely used. Developers could easily create some applications in HyperCard, but to create more complex applications, they had to switch to more difficult general-purpose development tools. The need for unification of these approaches was recognized early at Apple, leading to the formation of a research and development project to build a new development platform for the Mac, discussed in the next section. Looking forward, the rapid development capabilities pioneered by HyperCard were added to more sophisticated general-purpose development environments. This gradually reduced the need for systems like HyperCard, which was discontinued in 2004.

2.2 Family Farm

Many of the ideas that eventually emerged in AppleScript were initially explored as part of a research project code-named Family Farm, which was undertaken in the Advanced Technology Group (ATG) at Apple, starting in 1989. The research team was led by Larry Tesler and included Mike Farr, Mitchell Gass, Mike Gough, Jed Harris, Al Hoffman, Ruben Kleiman, Edmund Lai, and Frank Ludolph. Larry received a B.S. in Mathematics from Stanford University in 1965. In 1963, he founded and ran IPC, one of the first software development firms in Palo Alto, CA. From 1968-73, he did research at the Stanford A.I. Lab on cognitive modeling and natural language understanding, where he designed and developed PUB, one of the first markup languages with embedded tags and scripting. From 1973-80, he was a researcher at Xerox PARC, working on object-oriented languages, user interfaces, and desktop publishing. He joined Apple in 1980

to work on the Lisa. In 1986, he was named director of Advanced Development, and in 1987, the first VP of Advanced Technology, a new group focused on research.

The original goal of Family Farm was to create a new integrated development environment for the Macintosh OS. Family Farm included work on a new system-level programming language, an interprocess communication model, a user-level scripting language, and object component models, in addition to other areas.

The first AppleScript specification, which was the foundation for later development, was written by the Family Farm team. This document defined the basic approach to generalizing HyperTalk chunk expressions to create AppleScript object-specifiers and Apple Events (described in detail below). I believe credit for these ideas must be shared equally by the Family Farm team, which generated many ideas, and the teams which turned these ideas into usable systems.

As a research project that grew quickly, the organization put in place for Family Farm turned out not to be sufficient to build and deliver commercial products. The team was in the research group, not the system software group; No changes to the Macintosh system could be shipped to customers without approval of the system software group. And if Family Farm did get approval, they would have to follow the strict software development, scheduling, and quality control processes enforced by the system software group. Over time it became clear that the Family Farm team was also too small to achieve its vision.

After about a year and a half of work, the Family Farm project was disbanded, and new teams were created to design and implement some of the concepts investigated by Family Farm.

One of the main themes to emerge from Family Farm was a focus on techniques for *integrating* applications. As mentioned in Section 1.2, integrating graphical applications can be done at several levels: visual embedding, behavioral coordination, or data exchange. The spin-off projects were Apple Events, AppleScript, and OpenDoc. The Apple Events project, formed in mid-1990 from a subset of the Family Farm team, developed the underlying communication model on which later projects were based. Later projects involved larger teams that pulled key members from outside Family Farm. AppleScript was next, then OpenDoc, both within the Developer Tools Group at Apple. AppleScript focused on data and behavioral integration. The OpenDoc project, which is not discussed in detail here, focused on visual integration by embedding components. Family Farm's transition from research to product development was a difficult one; in the end the primary product transferred from Family Farm to its descendants was an inspiring vision.

2.3 Apple Event Manager

The Apple Event Manager provides an inter-application communication platform for the Macintosh. It was designed

with scripting in mind—however, the design was completed before development of AppleScript began. When I started at Apple in April 1991, my first job was to do a complete code review of Apple Events, which was nearing the end of its beta testing period. I sat in a conference room with Ed Lai for over a week reading the code line by line and also jumping around to check assumptions and review interrelationships. Ed was the primary developer of Apple Events code. The system was written in Pascal, as was most of the Macintosh system software of that era. The Apple Events team was part of the Developer Tools group and was originally managed by Larry Tesler (who was also still VP in ATG), but was later taken over by Kurt Piersol.

In designing Apple Events, Kurt, Ed and the team had to confront a serious limitation of the Macintosh OS: in 1990, the Macintosh OS could switch processes no more than 60 times a second. If a process performed only a few instructions before requesting a switch, the machine would idle until 1/60th of a second had elapsed. A fine-grained communication model, at the level of individual procedure or method calls between remote objects, would be far too slow: while a script within a single application could easily call thousands of methods in a fraction of a second, it would take several seconds to perform the same script if every method call required a remote message and process switch. As a result of this limitation of the OS, traditional remote procedure calls (RPC) could not be used. Fine-grained RPC was used in CORBA and COM, which were being developed at the same time.

The Macintosh OS needed a communication model that allowed objects in remote applications to be manipulated without requiring many process round-trips. The Apple Events communication model uses a generalization of HyperCard's chunk expressions. Just as a HyperCard command contains a verb and one or more chunk expressions in its predefined internal language, an Apple Event contains a verb and a set of chunk expressions that refer to objects and properties in the target application. The generalized chunk expressions are called *object specifiers*. Apple Events address the process-switching bottleneck by making it natural to pack more behavior into a single message, thereby reducing the need for communication round-trips between processes. An Apple Event is in effect a small query/update program that is formulated in one application and then sent to another application for interpretation.

Kurt Piersol and Mike Farr debated whether there should be few commands that could operate on many objects, or a large number of specific commands as in traditional command-line interfaces. For example, on Unix there are different commands to delete print jobs (lprm), directories (rmdir), and processes (kill). The analogy with verbs and nouns in English helped Kurt win the argument for few commands (verbs) that operate on general objects (nouns). For example, in AppleScript there is a single delete com-

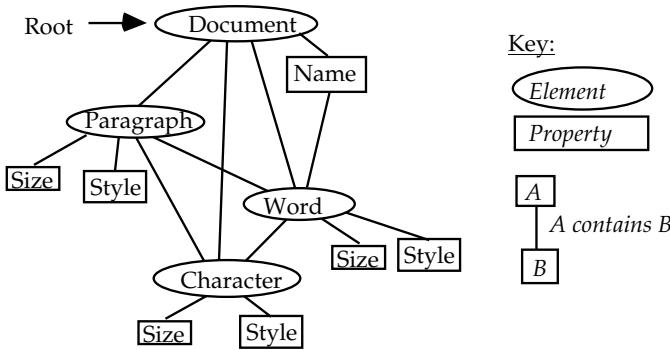


Figure 1. The properties and elements in a simple object model.

mand that can delete paragraphs or characters from a word-processing document, or files from the file system. Having a small number of generic verbs, including set, copy, delete, insert, open and close, proved to be more extensible.

2.3.1 Object Specifiers

Object specifiers are symbolic references to objects in an application. One application can create object specifiers that refer to objects in another application, called the *target*. The specifiers can then be included in an Apple Event and sent to the target application. These symbolic references are interpreted by the target application to locate the actual remote objects. The target application then performs the action specified by the verb in the Apple Event upon the objects.

For example, a single Apple Event can be used to copy a paragraph from one location to another in a document; the source and target location are included in the event as object specifiers. The content of the paragraph remains local in the target application. Traditional RPC models would require the client to retrieve the paragraph contents, and then send it back to the target as a separate insertion operation.

Object specifiers provide a view of application data that includes *elements* and *properties*. An element name represents a collection of values; a property has a single value. The value of a property may be either a basic value, for example an integer or a string, or another object. Elements are always objects.

A simple object model is illustrated in Figure 1. A document has multiple paragraph elements and a name property. A paragraph has style and size properties and contains word and character elements.

The distinction between properties and elements is related to the concept of cardinality in entity-relationship modeling [9] and UML class diagrams [32]. Cardinality indicates the maximum number of objects that may be involved in a relationship. The most important distinction is between single-valued (cardinality of 1) relationships and multivalued (cardinality greater than 1). The entity-relationship model also includes attributes, which identify scalar, prim-

itive data values. An AppleScript property is used for both attributes and single-valued relationships. Elements are used to describe multivalued relationships.

The name identifying a set of elements is called a *class* name, identifying a specific kind of contained object and/or its role. For example, a Family object might have elements parents and children, which are elements that refer to sets of Person objects. Object specifiers allow application-specific names for elements and properties, which generalize the fixed set of predefined names available in HyperCard.

Object specifiers also generalize HyperCard chunk expressions in other ways. One extension was the addition of *conditions* to select elements of a collection based on their properties. This extension made object specifiers a form of query language with significant expressive power.

Supporting Apple Events was frequently quite difficult. To create a new scriptable application, the software architect must design a scripting interface in addition to the traditional GUI interface. If the existing application architecture separates views (graphical presentations) from models (underlying information representation) [34], then adding scripting is usually possible. In this case the internal methods on the model may be exposed to scripts. There are two reasons why direct access to an existing object model may not be sufficient:

1. Users often want to control the user interface of an application, not just internal data of the application. Thus the scripting interface should provide access to both the view objects and the model objects.
2. The interfaces of the internal object model may not be suitable as an external scripting interface. In this case the scripting interface is usually implemented to provide another abstract view of the internal model.

Even with a good architecture, it can be difficult to retrofit an existing GUI application to include a second interface for scripting. If the application does not use a model-view architecture, then adding scripting is much more difficult.

The Apple Events team created a support library to assist application writers in interpreting object specifiers. It interprets nesting and conditions in the object specifiers, while using application callbacks to perform primitive property and element operations.

2.3.2 Apple Events Implementation

Object specifiers are represented in Apple Events as nested record structures, called *descriptors* [11]. Descriptors use a self-describing, tagged tree data structure designed to be easily transported or stored in a flattened binary format. Descriptors can either contain primitive data, a list of descriptors, or a labeled product of descriptors. Primitive data types include numbers (small and large integers and floats), pictures, styled and unstyled text, process IDs, files, and aliases. All the structures, types, and record fields are identified by

four-byte type codes. These codes are chosen to be human-readable to facilitate low-level debugging.

Each kind of object specifier is a record with fields for the class, property name, index, and container. The container is either another object specifier record or null, representing the default or root container of the application. Events may be sent synchronously or asynchronously. The default behavior of Apple Events is stateless—the server does not maintain state for each client session. However, Apple Events supports a simple form of transaction: multiple events can be tagged with a transaction ID, which requires an application to perform the events atomically or else signal an error.

Apple Events was first released with Macintosh System 7 in 1991. The entire Apple Events system was designed, implemented, and shipped in the Macintosh OS before any products using it were built. It was a complex problem: applications could not be built until the infrastructure existed, but the infrastructure could not be validated until many applications had used it. In the end, the Apple Events team adopted a “build it and they will come” approach. They designed the system as well as they could to meet predicted needs. Only a few small sample applications were developed to validate the model. In addition, the operating system team defined four standard events with a single simple parameter: open application, open documents, print documents, and quit. These first basic events did not use object specifiers; the open and print events used a vector of path names as arguments.

Later projects, including AppleScript, had to work around many of the Apple Events design choices that were made essentially within a vacuum. For example, Apple Events included some complex optimized message that were never used because they were too unwieldy. For example, if an array of values all has a common prefix, this prefix can be defined once and omitted in each element of the array. This was originally motivated by a desire to omit repetitive type information. This optimization is not used by AppleScript because it is difficult to detect when it could be used, and the reduction in message length provided by the optimization does not significantly affect performance.

3. The AppleScript Language

My primary task was to lead the design and implementation of the AppleScript language. After I decided to join Apple I mentioned the opportunity to Warren Harris. I enjoyed working with Warren at HP and thought he would be a great addition to the AppleScript effort. Warren has a BS and MS inree EE from the University of Kansas. At HP Warren was still working on his “Abel Project Posthumous Report”, which contained all the ideas we had discussed, but had not time to complete, while working together at HP Labs [25]. Warren talked to Kurt and eventually decided to join the AppleScript team as a software engineer. He quickly became the co-architect and primary implementor of the language.

3.1 Requirements

AppleScript is intended to be used by all users of the Macintosh OS. This does not imply that all users would use AppleScript to the same degree or in the same way—there is clearly a wide range of sophistication in users of the Macintosh OS, and many of them have no interest in, or need for, learning even the simplest form of programming language. However, these users could *invoke* scripts created by other users, so there were important issues of packaging of scripts, in addition to developing them. More sophisticated users might be able to *record* or *modify* a script even if they could not write it. Finally, it should be possible for non-computer specialists to *write* scripts after some study.

The language was primarily aimed at *casual* programmers, a group consisting of programmers from all experience levels. What distinguishes casual programming is that it is infrequent and in service of some larger goal—the programmer is trying to get something else done, not create a program. Even an experienced software developer can be a casual programmer in AppleScript.

The team recognized that scripting is a form of programming and requires more study and thought than using a graphical interface. The team felt that the language should be easy enough to learn and use to make it accessible to non-programmers, and that such users would learn enough programming as they went along to do the things that they needed to do.

Programs were planned to be a few hundred lines long at most, written by one programmer and maintained by a series of programmers over widely spaced intervals. Scripts embedded inside an application were also expected to be small. In this case, the application provides the structure to interconnect scripts; one script may send messages to an application object that contains another script. Because scripts are small, compilation speed was not a significant concern. Readability was important because it was one way to check that scripts did not contain malicious code.

In the early ’90s computer memory was still expensive enough that code size was a significant issue, so the AppleScript compiler and execution engine needed to be as small as possible. Portability was not an issue, since the language was specifically designed for the Macintosh OS. Performance of scripts within an application was not a significant concern, because complex operations like image filtering or database lookup would be performed by applications running native code, not by scripts. Due to the high latency of process switches needed for communication, optimization of communication costs was the priority.

3.2 Application Terminologies

An *application terminology* is a dictionary that defines the names of all the events, properties, and elements supported by an application. A terminology can define names as plural

```

reference ::=

  propertyName
  | 'beginning'
  | 'end'
  | 'before' term
  | 'after' term
  | 'some' singularClass
  | 'first' singularClass
  | 'last' singularClass
  | term ('st' | 'nd' | 'rd' | 'th') anyClass
  | 'middle' anyClass
  | plural ['from' term toOrThrough term]
  | anyClass term [toOrThrough term]
  | singularClass 'before' term
  | singularClass 'after' term
  | term ('of' | 'in' | 's') term
  | term ('whose' | 'where' | 'that') term

plural ::= pluralClass | 'every' anyClass
toOrThrough ::= 'to' | 'thru' | 'through'

call ::= message '(' expr* ')'
       | message ['in' | 'of'] [term] arguments
       | term name arguments

message ::= name | terminologyMessage

arguments ::= (preposition expression | flag | record)*

flag ::= ('with' | 'without') [name]+
record ::= 'given' (name ':' expr)*
preposition ::=
  | 'to' | 'from' | 'thru' | 'through'
  | 'by' | 'on' | 'into' | terminologyPreposition

```

Figure 2. AppleScript grammar for object references and message sends.

or masculine/feminine, and this information can be used by the custom parser for a dialect.

One of the main functions of AppleScript is to send and receive Apple Events (defined in Section 2.3). Sending an Apple Event is analogous to an object-oriented message send, while handling an Apple Event is done by defining a method whose name is an Apple Event. One view of this arrangement is that each application is an object that responds to a variety of messages containing complex argument types. AppleScript encourages a higher-level view in which each application manages a set of objects to which messages may be sent.

AppleScript also provides special syntax for manipulating Apple Event object specifiers, which are called “object references” in AppleScript documentation. When an operation is performed on an object reference, an Apple Event is created containing the appropriate verb and object specifier. Thus AppleScript needed to provide a concise syntax for expressing Apple Events. These AppleScript expressions create object specifiers:

```

the first word of paragraph 22
name of every figure of document "taxes"
the modification date of every file whose size > 1024

```

The first example is a reference to a particular word of a paragraph. The second refers to the collection of names associated with all figures in a document named “taxes”. The last one refers to the modification dates of large files.

For example, the object reference `name of window 1` identifies the name of the first window in an application. Object references are like first-class pointers that can be dereferenced or updated. An object reference is automatically dereferenced when a primitive value is needed:

```
print the name of window 1
```

The primitive value associated with an object reference can be updated:

```
set the name of window 1 to "Taxes"
```

These examples illustrate that object specifiers can act as both l-values and r-values.

Figure 2 includes the part of the AppleScript grammar relating to object specifiers. The grammar makes use of four nonterminals that represent symbols from the application terminology: for `propertyName`, `singularClass`, `pluralClass`, and `anyClass`. As mentioned in Section 2.3.1, a terminology has properties and elements, which are identified by a class name. Property names are identified by lexical analysis and passed to the parser. For class names the terminology can include both plural and singular forms or a generic “any” form. For example, `name` is a property, `window` is a singular class, and `windows` is a plural class. The grammar then accepts `windows from 1 to 10` and `every window from 1 to 10`,

Figure 2 also summarizes the syntax of messages. Arguments can be given by position or name after the `given` keyword. In addition, a set of standard prepositions can be used as argument labels. This allows messages of the form:

```
copy paragraph 1 to end of document
```

The first parameter is `paragraph 1`, and the second argument is a prepositional argument named `to` with value `end of document`.

One of the most difficult aspects of the language design arose from the fundamental ambiguity of object references: an object reference is itself a first-class value, but it also denotes a particular object (or set of objects) within an application. In a sense an object reference is like a symbolic pointer, which can be dereferenced to obtain its value; the referenced value can also be updated or assigned through the object reference. The difficulties arose because of a desire to hide the distinction between a reference and its value. The solution to this problem was to dereference them automatically when necessary, and require special syntax to create an object reference instead of accessing its value. The expression `a reference to o` creates a first-class reference to an object

described by o. Although the automatic dereferencing handles most cases, a script can explicitly dereference r using the expression value **of** r. The examples above can be expressed using a reference value:

```
set x to a reference to the name of window 1
```

The variable x can then be used in place of the original reference. The following statements illustrate the effect of operations involving x:

```
print the value of x
print x
set the value of x to "Taxes"
set x to "Taxes"
```

The first and second statements both print the name of the window. In the first statement the dereference is explicit, but in the second it happens implicitly because print expects a string, not a reference. The third statement changes the name of the window, while the last one changes the values of the variable x but leaves the window unchanged.

An object reference can be used as a base for further object specifications.

```
set x to a reference to window 1
print the name of x
```

Figure 3 and 4 describe the custom terminology dictionary for iChat, a chat, or instant messaging, program for the Macintosh. It illustrates the use of classes, elements, properties, and custom events. Terminologies for large applications are quite extensive: the Microsoft Word 2004 AppleScript Reference is 529 pages long, and the Adobe Photoshop CS2 AppleScript Scripting Reference is 251 pages. They each have dozens of classes and hundreds of properties.

In addition to the terminology interpreted by individual applications, AppleScript has its own terminology to identify applications on multiple machines. The expression application "name" identifies an application. The expression application "appName" **of** machine "machineName" refers to an application running on a specific machine. A block of commands can be targeted at an application using the **tell** statement:

```
tell application "Excel" on machine x
put 3.14 into cell 1 of row 2 of window 1
end
```

This is an example of a static **tell** command, because the name of the target application is known statically, at compile time. The target of the **tell** statement can also be a dynamic value rather than an application literal. In this case the terminology is not loaded from the application. To communicate with a dynamic application using a statically specified terminology, a dynamic **tell** can be nested inside a static **tell**; the outer one sets the static terminology, while the inner one defines the dynamic target. This brings up the possibility that applications may receive messages that they

Class application: iChat application

<i>Plural form:</i>	applications
<i>Elements:</i>	account, service , window, document
<i>Properties:</i>	
idle time	integer
	<i>Time in seconds that I have been idle.</i>
image	picture
	<i>My image as it appears in all services.</i>
status message	string
	<i>My status message, visible to other people while I am online.</i>
status	string
	<i>My status on all services: away/offline/available.</i>

Class service: An instant-messaging service

<i>Plural form:</i>	services
<i>Elements:</i>	account
<i>Properties:</i>	
status	string
	<i>The status of the service.: disconnecting/connected/connecting/disconnected.</i>
id	string
	<i>The unique id of the service.</i>
name	string
	<i>The name of the service.</i>
image	picture
	<i>The image for the service.</i>

Class account: An account on a service

<i>Plural form:</i>	accounts
<i>Properties:</i>	
status	string
	<i>away/offline/available/idle/unknown.</i>
id	string
	<i>The account's service and handle. For example: AIM:JohnDoe007.</i>
handle	string
	<i>The account's online name.</i>
name	string
	<i>The account's name as it appears in the buddy list.</i>
status message	
	<i>The account's status message.</i>
capabilities	list
	<i>The account's messaging capabilities.</i>
image	picture
	<i>The account's custom image.</i>
idle time	integer
	<i>The time in seconds the account has been idle.</i>

Figure 3. iChat Suite: Classes in the iChat scripting terminology [13].

Events

log in service

Log in a service with an account. If the account password is not in the keychain the user will be prompted to enter one.

log out service

Logs out of a service, or all services if none is specified.

send message to account

Send account a text message or video invitation.

Figure 4. iChat Suite: Events in the iChat scripting terminology [13].

do not understand. In such situations, the application should return an error.

Integration of multiple applications opens the possibility that a single command may involve object references from multiple applications. The target of a message is determined by examining the arguments of the message. If all the arguments are references to the same application, then that application is the target. But if the arguments contain references to different applications, one of the applications must be chosen as the target. Since applications can interpret only their own object specifiers, the other object references must be evaluated to primitive values, which can then be sent to the target application.

```
copy the name of the first window
    of application "Excel"
    to the end of the first paragraph
        of app "Scriptable Text Editor"
```

This example illustrates copying between applications without using the global clipboard. AppleScript picks the target for an event by examining the first object reference in the argument list. If the argument list contains references to other applications, the values of the references are retrieved and passed in place of the references in the argument list.

Standard events and reference structures are defined in the *Apple Event Registry*. The Registry is divided into suites that apply to domains of application. Suites contain specifications for classes and their properties, and events. Currently there are suites for core operations, text manipulation, databases, graphics, collaboration, and word services (spell-checking, etc.).

Jon Pugh, with a BSCS from Western Washington University in 1983, was in charge of the Apple Events registry. He also helped out with quality assurance and evangelism. Since then he has worked on numerous scriptable applications, and created “Jon’s Commands,” a shareware library of AppleScript extensions.

Terminologies also provide natural-language names for the four-letter codes used within an Apple Event. This metadata is stored in an application as a resource. As discussed in

Section 3.4 below, the terminology resources in an application are used when parsing scripts targeting that application.

3.3 Programming Language Features

AppleScript’s programming language features include variables and assignment, control flow, and basic data structures. Control flow constructs include conditionals, a variety of looping constructs, subroutines, and exceptions. Subroutines allow positional, prepositional, and keyword parameters. Data structures include records, lists, and objects. Destructuring bind, also known as pattern matching, can be used to break apart basic data structures. Lists and records are mutable. AppleScript also supports objects and a simple transaction mechanism.

AppleScript has a simple object model. A *script object* contains *properties* and *methods*. Methods are dynamically dispatched, so script objects support a simple form of object-oriented programming. The following simple script declaration binds the name Counter to a new script object representing a counter:

```
script Counter
    property count : 0
    to increment
        set count to count + 1
        return count
    end increment
end script
```

A script declaration can be placed inside a method to create multiple instances. Such a method is called a factory method, and is similar to a constructor method in Java. Since script can access any lexically enclosing variables, all the objects created by a factory have access to the state of the object that constructed them. The resulting pattern of object references resembles the class/metaclass system in Smalltalk [23], although in much simpler form.

AppleScript’s object model is a prototype model similar to that employed by Self [37], as opposed to the container-based inheritance model of HyperTalk. Script objects support single inheritance by delegating unhandled commands to the value in their parent property [36]. JavaScript later adopted a model similar to AppleScript.

The top level of every script is an implicit object declaration. Top-level properties are *persistent*, in that changes to properties are saved when the application running the script quits. A standalone script can be stored in a script file and executed by opening it in the Finder. Such a script can direct other applications to perform useful functions, but may also call other script files. Thus, script files provide a simple mechanism for modularity.

AppleScript provides no explicit support for threading or synchronization. However, the application hosting a script can invoke scripts from multiple threads: the execution engine was thread-safe when run by the non-preemptive scheduling in the original Macintosh OS. It is not safe when

English	the first character of every word whose style is bold
Japanese	スタイル = ボールド である すべての 単語 の 最初 の 文字
French	le premier caractère de tous les mots dont style est gras
Professional	{ words style == bold }.character[1]

Figure 5. Illustration of dialects.

run on multiple preemptive threads on Mac OS X. Script objects can also be sent to another machine, making mobile code possible.

3.4 Parsing and Internationalization

The AppleScript parser integrates the terminology of applications with its built-in language constructs. For example, when targeting the Microsoft Excel™ application, spreadsheet terms are known by the parser—nouns like `cell` and `formula`, and verbs like `recalculate`. The statement `tell application "Excel"` introduces a block in which the Excel terminology is available. The terminology can contain names that are formed from multiple words; this means that the lexical analyzer must be able to recognize multiple words as a single logical identifier. As a result, lexical analysis depends upon the state of the parser: on entering a tell block, the lexical analysis tables are modified with new token definitions. The tables are reset when the parser reaches the end of the block. This approach increases flexibility but makes parsing more difficult. I believe the added complexity in lexing/parsing makes it more difficult for users to write scripts.

Apple also required that AppleScript, like most of its other products, support localization, so that scripts could be read and written in languages other than English. Scripts are stored in a language-independent internal representation. A *dialect* defines a presentation for the internal language. Dialects contain lexing and parsing tables, and printing routines. A script can be presented using any dialect—so a script written using the English dialect can be viewed in Japanese. Examples are given in Figure 5. For complete localization, the application terminologies must also include entries for multiple languages. Apple developed dialects for Japanese and French. A “professional” dialect, which resembles Java, was created but not released.

There are numerous difficulties in parsing a programming language that resembles a natural language. For example, Japanese does not have explicit separation between words. This is not a problem for language keywords and names from the terminology, but special conventions were required to recognize user-defined identifiers. Other languages have complex conjugation and agreement rules that are difficult to implement. Nonetheless, the internal representation of AppleScript and the terminology resources contain information to support these features.

The AppleScript parser was created using Yacc [29], a popular LALR parser generator. Poor error messages are a

common problem with LALR parsing [1]. I wrote a tool that produces somewhat better error messages by including a simplified version of the follow set at the point where the error occurred. The follow set was simplified by replacing some common sets of symbols (like binary operators) with a generic name, so that the error message would be “expected binary operator” instead of a list of every binary operator symbol. Despite these improvements, obscure error messages continue to be one of the biggest impediments to using AppleScript.

3.5 AppleScript Implementation

During the design of AppleScript in mid-1991, we considered building AppleScript on top of an existing language or runtime. We evaluated Macintosh Common Lisp (MCL), Franz Lisp, and Smalltalk systems from ParcPlace and Digitalk. These were all good systems, but were not suitable as a foundation for AppleScript for the same reason: there was not sufficient separation between the development environment and the runtime environment. Separating development from execution is useful because it allows compiled script to be executed in a limited runtime environment with low overhead. The full environment would be needed only when compiling or debugging a script.

Instead, we developed our own runtime and compiler. The runtime includes a garbage collector and byte-code interpreter. The compiler and runtime were loaded separately to minimize memory footprint.

One AppleScript T-shirt had the slogan “We don’t patch out the universe”. Many projects at Apple were implemented by “patching”: installing new functions in place of kernel operating system functions. The operating system had no protection mechanisms, so any function could be patched. Patches typically had to be installed in a particular order, or else they would not function. In addition, a bug in a patch could cause havoc for a wide range of applications.

AppleScript did not patch any operating system functions. Instead the system was carefully packaged as a thread-safe QuickTime component. QuickTime components are a lightweight dynamic library mechanism introduced by the QuickTime team. Only one copy of the AppleScript compiler and runtime was loaded and shared by all applications on a machine. The careful packaging is one of the reasons AppleScript was able to continue running unchanged through numerous operating system upgrades, and even onto the PowerPC.

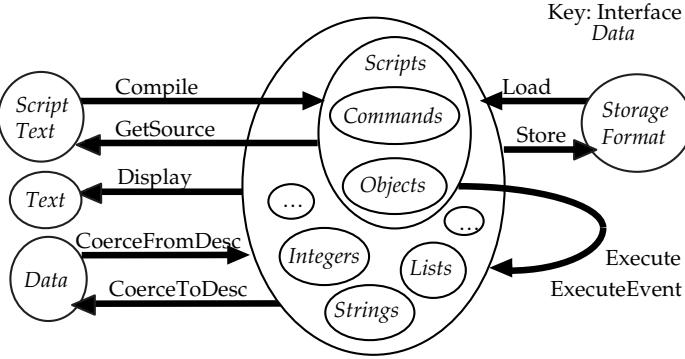


Figure 6. Overview of the Open Scripting API.

The AppleScript runtime is implemented in C++. The entire system, including dialects, is 118K lines of code, including comments and header files. Compiling the entire AppleScript runtime took over an hour on the machines used by the development team. After the alpha milestone, the development team was not allowed to produce official builds for the testing team. Instead, the code had to be checked in and built by a separate group on a clean development environment. This process typically took 8 to 12 hours, because the process was not fully automated, so there was sometimes a significant lag between identifying a bug and delivering a fix to the quality assurance team. This was a significant source of frustration for the overall team.

4. Script Management

Open Scripting Architecture (OSA) allows any application to manipulate and execute scripts [11]. The Open Scripting API is centered around the notion of a *script*, as shown in Figure 6. A script is either a data value or a program. Many of the routines in the API are for translating between scripts and various external formats. Compile parses script text and creates a script object, while GetSource translates a script object back into human-readable script text. Display translates a value into a printed representation. When applied to a string, e.g. “Gustav”, GetSource returns a program literal “Gustav”, while Display just returns the text Gustav. CoerceFromDesc and CoerceToDesc convert AppleScript values to and from Apple Event descriptors. Load and Store convert to/from compact binary byte-streams that can be included in a file.

The Execute function runs a script in a context. A context is a script that contains bindings for global variables.

At its simplest, the script management API supports the construction of a basic script editor that can save scripts as stand-alone script applications.

The OSA API does not include support for debugging, although this was frequently discussed by the team. However, other companies have worked around this problem and created effective debugging tools (Section 6.3).

4.1 Embedding

The script management API also supports attaching scripts to objects in an existing application. Such scripts can be triggered during normal use of the application. This usage is supported by the ExecuteEvent function, which takes as input a script and an Apple Event. The event is interpreted as a method call to the script. The corresponding method declaration in the script is invoked. In this way an application can pass Apple Events to scripts that are attached to application objects.

Embedded scripts allow default application behavior to be customized, extended, or even replaced. For example, the Finder can run a script whenever a file is added to a folder, or an email package can run a script when new mail is received. Scripts can also be attached to new or existing menu items to add new functionality to an application. By embedding a universal scripting language, application developers do not need to build proprietary scripting languages, and users do not need to learn multiple languages. Users can also access multiple applications from a single script. AppleScript demonstrated the idea that a single scripting language could be used for all applications, while allowing application-specific behaviors to be incorporated so that the language was specialized for use in each application.

Embedding can also be used to create entire applications. In this case there is no predefined application structure to which scripts are attached. Instead, the user builds the application objects — for data and user interfaces, and then attaches scripts to them. Several application development tools based on AppleScript are described in Section 6.

4.2 Multiple Scripting Languages

Halfway through the development of AppleScript, Apple management decided to allow third-party scripting languages to be used in addition to AppleScript. A new API for managing scripts and scripting language runtime engines had to be designed and implemented. These changes contributed to delays in shipping AppleScript. However, they also led to a more robust architecture for embedding.

In February of 1992, just before the first AppleScript alpha release, Dave Winer convinced Apple management that having one scripting language would not be good for the Macintosh. At that time, Dave Winer was an experienced Macintosh developer, having created one of the first outliner applications, ThinkTank. In the early 1990s, Dave created an alternative scripting system, called Frontier. Before I joined the project, Apple had discussed the possibility of buying Frontier and using it instead of creating its own language. For some reason the deal fell through, but Dave continued developing Frontier. Apple does not like to take business away from developers, so when Dave complained that the impending release of AppleScript was interfering with his product, Apple decided the AppleScript should be opened up to multiple scripting languages. The AppleScript team mod-

ified the OSA APIs so that they could be implemented by multiple scripting systems, not just AppleScript. As a result, OSA is a generic interface between clients of scripting services and scripting systems that support a scripting language. Each script is tagged with the scripting system that created it, so clients can handle multiple kinds of script without knowing which scripting system they belong to.

Dave Winer's Frontier is a complete scripting and application development environment that eventually became available as an Open Scripting component. Dave went on to participate in the design of web services and SOAP [4]. Tcl, JavaScript, Python and Perl have also been packaged as Open Scripting components.

4.3 Recording Events as Scripts

The AppleScript infrastructure supports recording of events in *recordable* applications, which publish events in response to user actions. Donn Denman, a senior software engineer on the AppleScript team with a BS in Computer Science and Math from Antioch College, designed and implemented much of the infrastructure for recording. At Apple he worked on Basic interpreters. He was involved in some of the early discussions of AppleScript, worked on Apple Events and application terminologies in AppleScript. In addition, Donn created MacroMaker, a low-level event record and playback system for Macintosh OS System 5. Working on MacroMaker gave Donn a lot of experience in how recording should work.

Recording allows automatic generation of scripts for repeated playback of what would otherwise be repetitive tasks. Recorded scripts can be subsequently generalized by users for more flexibility. This approach to scripting alleviates the “staring at a blank page” syndrome that can be so crippling to new scripters. Recording is also useful for learning the terminology of basic operations of an application, because it helps users to connect actions in the graphical interface with their symbolic expression in script.

Recording high-level events is different from recording low-level events of the graphical user interface. Low-level events include mouse and keyboard events. Low-level events can also express user interface actions, e.g. “perform Open menu item in the File menu”, although the response to this event is usually to display a dialog box, not to open a particular file. Additional low-level events are required to manipulate dialog boxes by clicking on interface elements and buttons. Low-level events do not necessarily have the same effect if played back on a different machine, or when different windows are open. High-level events are more robust because they express the intent of an action more directly. For example, a high-level event can indicate which file to open.

Recording is supported by a special mode in the Apple Event manager, based on the idea that a user’s actions in manipulating a GUI interface can be described by a corresponding Apple Event. For example, if the user selects the

File Open menu, then finds and selects a file named “Resumé” in a folder named “Personal”, the corresponding Apple Event would be a FileOpen event containing the path “Personal:Resumé”. To be recordable, an application must post Apple Events that describe the actions a user performs with the GUI.

Recordable applications can be difficult to build, since they must post an Apple Event describing each operation performed by a user. The AppleScript team promoted an architecture that turned this difficulty into a feature. We advocated that applications should be factored into two parts, a GUI and a back end, where the only communication from the GUI to the back end is via Apple Events. With this architecture, all the core functionality of the application must be exposed via Apple Events, so the application is inherently scriptable. The GUI’s job becomes one of translating low-level user input events (keystrokes and mouse movements) into high-level Apple Events. An application built in this way is inherently recordable; the Apple Event manager simply records the Apple Events that pass from the GUI to the back end. If an application is already scriptable, it can be made recordable by arranging for the user interface to communicate with the application model only through Apple Events.

The reality of recording is more complex, however. If there is a type Apple Event to add characters into a document, the GUI must forward each character immediately to the back end so that the user will see the result of typing. During recording, if the user types “Hello” the actions will record an undesirable script:

```
type "H"  
type "e"  
type "I"  
type "l"  
type "o"
```

It would be better to record type “Hello”. To get this effect, the GUI developer could buffer the typing and send a single event. But then the user will not see the effect of typing immediately. AppleEvents has the ability to specify certain events as *record-only*, meaning that it is a summary of a user’s actions and should not be executed. Creating such summaries makes developing a recordable application quite difficult.

In 2006 twenty-five recordable applications were listed on Apple’s website and in the AppleScript Sourcebook [8], one of several repositories of information about AppleScript. Some, but fewer than half, of the major productivity applications are recordable. Recordable applications include Microsoft Word and Excel, Netscape Navigator, Quark Express (via a plugin) and CorelDRAW.

One of the inherent difficulties of recording is the ambiguity of object specification. As the language of events becomes more rich, there may be many ways to describe a given user action. Each version might be appropriate for a

given situation, but the system cannot pick the correct action without knowing the intent of the user. For example, when closing a window, is the user closing the front window or the window specifically named “Example”? This is a well-known problem in research on programming by example, where multiple examples of a given action can be used to disambiguate the user’s intent. Allen Cypher did fundamental research on this problem while at Apple. He built a prototype system called Eager that anticipated user actions by watching previous actions [21, 22]. AppleScript does not have built-in support for analyzing multiple examples. There are also ambiguities when recording and embedding are combined: if a recorded event causes a script to execute, should the original event or the events generated by the script be recorded? Application designers must decide what is most appropriate for a given situation. Cypher worked with Dave Curbow in writing guidelines to help developers make these difficult choices [26].

Recording can also be used for other purposes. For example, a help system can include step-by-step instructions defined by a script. The steps can be played as normal scripts, or the user can be given the option of performing the steps manually under the supervision of the help system. By recording the user’s actions, the help system can provide feedback or warnings when the user’s actions do not correspond to the script.

5. Development Process

Unlike most programming languages, AppleScript was designed within a commercial software development project. The team members are listed in Figure 7. AppleScript was designed by neither an individual nor a committee; the team used a collaborative design process, with significant user testing and evaluation. The project leaders guided the process and made final decisions: there was lively debate within the group about how things should work. The extended team included project management, user interface design and testing, documentation, and product marketing.

The AppleScript project had a strong quality assurance (QA) team. They created a large test suite which was run against nightly builds. From a management viewpoint, the QA group also had significant control over the project, because they were required to give final approval of a release.

The project was code-named “Gustav” after Donn’s massive Rottweiler dog. The dog slimed everything it came in contact with, and was the impetus behind a T-shirt that read “Script Happens”. The project tag line was “Pure Guava” because Gary Bond was designing a t-shirt that said “AppleScript: Pure Gold” and Warren Harris got him to change it to Pure Guava after the Ween album he was in love with at the time.

AppleScript and the associated tools were designed and implemented between 1990 and 1993. Figure 8 gives a timeline of the development process. The line labeled “changes”

Jens Alfke	Developer	Ron Karr	QA, Apple Events Developer
Greg Anderson	Developer, Scriptable Finder	Edmund Lai	Developer, Apple Events
Mike Askins	Engineering Project Manager	Ron Lichty	Manager, Finder
Gary Bond	QA	Bennet Marks	Developer
Scott Bongiorno	QA, User Testing	Mark Minshull	Manager
B. Bruce Brinson	Developer	Kazuhis Ohta	Developer, Dialects
Kevin Calhoun	Manager	Donald Olson	QA, Manager
Jennifer Chaffee	User Interface Design	Chuck Piercy	Marketing
Dan Clifford	Developer	Kurt Piersol	Architect
William Cook	Architect, Developer, Manager	James Redfern	QA, Developer
Sean Cotter	Documentation	Brett Sher	Developer, QA
Dave Curbow	User Interface Design	Laile Di Silvestro	QA, Developer
Donn Denman	Developer	Sal Soghoian	Product Manager
Sue Dumont	Developer, QA	Francis Stanbach	Developer, Scriptable Finder
Mike Farr	Marketing	Kazuhiko Tateda	Japanese Dialect
Mitch Gass	Documentation	Larry Tesler	Manager, VP
Laura Clark Hamersley	Marketing	Mark Thomas	Evangelist
Warren Harris	Architect, Developer	Susan Watkins	Marketing
Eric House	QA, Developer		

Figure 7. AppleScript and related project team members.

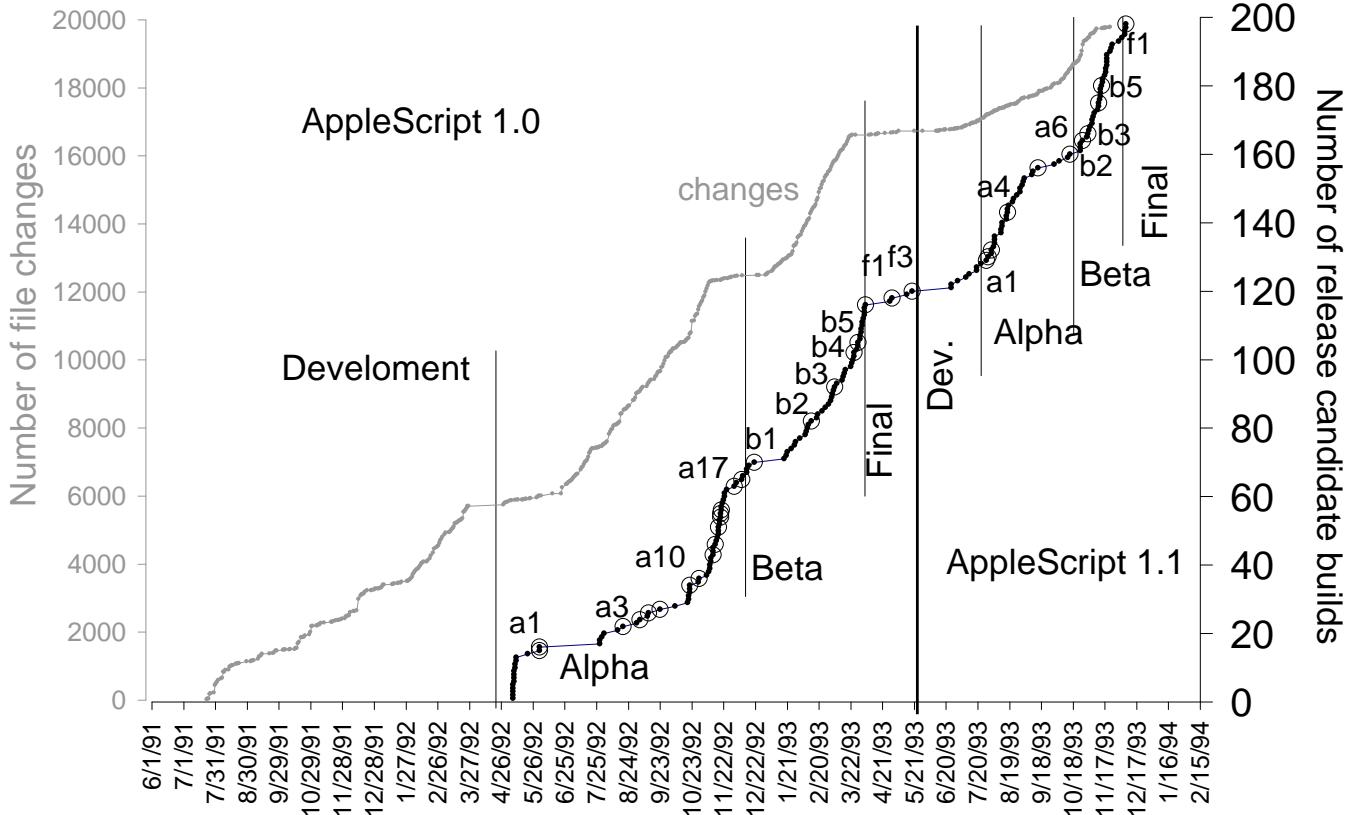


Figure 8. Development statistics: number of file changes and candidate builds.

shows the cumulative number of files changed during development (the scale is on the left). The second line shows the cumulative number of candidate release builds. The final candidate builds were created by Apple source control group from a clean set of sources and then given to the testing team. By placing source code control between development and testing, Apple ensured that each build could be recreated on a clean development environment with archived sources. Note that the number of file changes in the alpha or beta phases starts off slowly, then increases until just before the next milestone, when changes are no longer allowed unless absolutely necessary.

The AppleScript Beta was delivered in September 1992. In April 1993 the AppleScript 1.0 Developer's Toolkit shipped, including interface declaration files (header files), sample code, sample applications and the Scripting Language Guide.

The first end-user version, AppleScript 1.1, was released in September 1993 and included with System 7 Pro. In December 1993, the 1.1 Developer's Toolkit and Scripting Kit versions both released. In 1994, AppleScript was included as part of System 7.5.

In January 1993, Apple management decided that the next version of AppleScript had to have more features than AppleScript 1.1, but that the development must be done with half the number of people. Since this was not likely

to lead to a successful development process, Warren and I decided to leave Apple. Without leadership, the AppleScript group was disbanded. Many of the team members, including Jens Alfke, Donn Denman, and Donald Olson, joined Kurt Piersol on the OpenDoc team, which was working on visual integration of applications. AppleScript was integrated into the OpenDoc framework.

5.1 Documentation

Internal documentation was ad hoc. The team made extensive use of an early collaborative document management/writing tool called Instant Update, that was used in a wiki-like fashion, a living document constantly updated with the current design. Instant Update provides a shared space of multiple documents that were viewed and edited simultaneously by any number of users. Each user's text was color-coded and time-stamped. I have not been able to recover a copy of this collection of documents.

No formal semantics was created for the language, despite the fact that my PhD research and work at HP Labs was focused entirely on formal semantics of programming languages. One reason was that only one person on the team was familiar with formal semantics techniques, so writing a formal semantics would not be an effective means of communication. In addition, there wasn't much point in developing a formal semantics for the well-known features (objects,

inheritance, lexical closures, etc.), because the goal for this aspect of the language was to apply well-known constructs, not define new ones. There was no standard formal semantic framework for the novel aspects of AppleScript, especially the notion of references for access to external objects residing in an application. The project did not have the luxury of undertaking extensive research into semantic foundations; its charter was to develop and ship a practical language in a short amount of time. Sketches of a formal semantics were developed, but the primary guidance for language design came from solving practical problems and from user studies, rather than from a priori formal analysis.

The public documentation was developed by professional writers who worked closely with the team throughout the project. The primary document is *Inside Macintosh: Inter-application Communication*, which includes details on the Apple Event Manager and Scripting Components [11]. The AppleScript language is also thoroughly documented [2], and numerous third-party books have been written about it, for examples see [31, 24]. Mitch Gass and Sean Cotter documented Apple Events and AppleScript for external use. Mitch has a bachelor's degrees in comparative literature and computer science, and worked at Tandem and Amiga before joining Apple. Mitch worked during the entire project to provide that documentation, and in the process managed to be a significant communication point for the entire team.

5.2 User Testing

Following Apple's standard practice, we user-tested the language in a variety of ways. We identified novice users and asked them, "What do you think this script does?" The following questions illustrate the kinds of questions asked during user testing.

Part I. Please answer the following multiple choice questions about AppleScript.

3. Given the handler:

```
on doit from x to y with z
    return (x * y) + z
end doit
```

What does the following statement evaluate to?

```
doit with 3 from 8 to 5
```

- a) 29
- b) 43
- c) error
- d) other:

Part II. Please state which of the following AppleScript statements you prefer.

8. a) **put** "a", {"b", "c"} **into** x
- b) **put** {"a", {"b", "c"}} **into** x
9. a) window named "fred"
- b) window "fred"
10. a) window 1
- b) window #1
11. a) word 34
- b) word #34
12. a) "apple" < "betty"
- b) "apple" comes **before** "betty"

Part III. This section shows sequences of commands and then asks questions about various variables after they are executed.

15. Given the commands:

```
put {1, 2, 3} into x
put x into y
put 4 into item 1 of x
```

What is x?

- a) {1, 2, 3}
- b) {4, 2, 3}
- c) error
- d) other:

What is y?

- a) {1, 2, 3}
- b) {4, 2, 3}
- c) error
- d) other:

Part IV. In this section, assume that all AppleScript statements refer to window 1, which contains the following text:

```
this is a test
of the emergency broadcast system
```

18. What does the following statement evaluate to?

```
count every line of window 1
```

- a) 2
- b) 4, 5
- c) 9
- d) 14, 33
- e) 47
- f) error
- g) other:

What does the following statement evaluate to?

count each line **of** window 1

- a) 2
- b) 4, 5
- c) 9
- d) 14, 33
- e) 47
- f) error
- g) other:

21. What does the following statement evaluate to?

every line **of** window 1
whose first character = "x"

- a) {}
- b) error
- c) other:

One result of user testing concerned the choice of verb for assignment commands. The average user thought that after the command **put** x **into** y the variable x no longer retained its old value. The language was changed to use **copy** x **into** y instead. We also conducted interviews and a round-table discussion about what kind of functionality users would like to see in the system. In the summer of 1992, Apple briefed 50 key developers and collected reactions. The user interface team conducted controlled experiments of the usability of the language in early 1993, but since these took place during the beta-testing period, they were too late in the product development cycle to have fundamental impact.

6. AppleScript Applications and Tools

Much of the practical power of AppleScript comes from the applications and tools that work with scripts and handle events. From the viewpoint of AppleScript, applications are large, well-designed and internally consistent libraries of specialized functionality and algorithms. So, when used with a database application, AppleScript can perform data-oriented operations. When used with a text layout application, AppleScript can automate typesetting processes. When used with a photo editing application, AppleScript can perform complex image manipulation.

Since new libraries can be created to cover any application domain, only the most basic data types were supported in AppleScript directly. For example, string handling was minimal in AppleScript. AppleScript's capabilities were initially limited by the availability of scriptable applications. Success of the project required that many applications and diverse parts of the operating system be updated to support scripting.

A second benefit of pervasive scripting is that it can be used to provide a uniform interface to the operating system. With Unix, access to information in a machine is idiosyncratic, in the sense that one program was used to list print jobs, another to list users, another for files, and another for hardware configuration. I envisioned a way in which all these different kinds of information could be referenced uniformly.

A *uniform naming model* allows every piece of information anywhere in the system, be it an application or the operating system, to be accessed and updated uniformly. Application-specific terminologies allow applications to be accessed uniformly; an operating system terminology would provide access to printer queues, processor attributes, or network configurations. Thus, the language must support multiple terminologies simultaneously so that a single script can access objects from multiple applications and the operating system at the same time.

6.1 Scriptable Finder

Having a scriptable Finder was a critical requirement for AppleScript, since the Finder provides access to most system resources. However, it was difficult to coordinate schedules and priorities because the Finder and AppleScript teams were in different divisions within Apple. The Finder team was also pulled in many directions at once.

As a result, Finder was not fully scriptable when AppleScript shipped in 1992. The Finder team created a separate library, called the "Finder scripting extension", to provide some additional Finder script commands. The Finder had been rewritten in C++ from the ground up for System 7 to be extensible. But extensions relied on internal C++ dispatch tables, so the Finder was not dynamically extensible: it had to be recompiled for each extension. The Finder extension mechanism had been designed so that Finder functionality could grow incrementally. It was the mechanism for adding large quantities of new functionality to support a specific project.

It was not until 1997 that a scriptable Finder was released. A year later the Finder supported embedding, which greatly increased its power. Embedding allowed scripts to be triggered from within the Finder in response to events, for example opening a folder or emptying the trash.

6.2 Publishing Workflow

Automation of publishing workflows is a good illustration of AppleScript and scriptable applications. Consider the automation of a catalog publishing system. An office-products company keeps all its product information in a FileMaker Pro™ database that includes descriptions, prices, special offer information, and a product code. The product code identifies a picture of the product in a Kudos™ image database. The final catalog is a QuarkXPress™ document that is ready for printing. Previously, the catalog was produced manually,

a task that took a team of twenty up to a month for a single catalog.

An AppleScript script automates the entire process. The script reads the price and descriptive text from the FileMaker Pro database and inserts it into appropriate QuarkXPress fields. The script applies special formatting: it deletes the decimal point in the prices and superscripts the cents (e.g. 34⁹⁹). To make the text fit precisely in the width of the enclosing box, the script computes a fractional expansion factor for the text by dividing the width of the box by the width of the text (this task was previously done with a calculator). It adjusts colors and sets the first line of the description in boldface type. Finally, it adds special markers like “Buy 2 get 1 free” and “Sale price \$17⁹⁹” where specified by the database.

Once this process is automated, one person can produce the entire catalog in under a day, a tiny fraction of the time taken by the manual process. It also reduced errors during copying and formatting. Of course, creating and maintaining the scripts takes time, but the overall time is significantly reduced over the long run.

6.3 Scripting Tools

AppleScript included a simple and elegant script editor created by Jens Alfke, who had graduated from Caltech and worked with Kurt Piersol at Xerox Parc on Smalltalk-80 applications. Jens was one of the key developers on the AppleScript team; he focused on tools, consistency of the APIs and usability of the overall system.

Soon after AppleScript was released, more powerful script development environments were created outside Apple. They addressed one of the major weaknesses of AppleScript: lack of support for debugging. One developer outside Apple who took on this challenge is Cal Simone, who has also been an unofficial evangelist for AppleScript since its inception. Cal created *Scripter*, which allows users to single-step through a script. It works by breaking a script up into individual lines that are compiled and executed separately. The enclosing **tell** statements are preserved around each line as it is executed. Scripter also allows inspection of local variables and execution of immediate commands within the context of the suspended script. *Script Debugger* uses a different technique: it adds a special Apple Event between each line of a script. The Apple Event is caught by the debugger and the processing of the script is suspended. The current values of variables can then be inspected. To continue the script, the debugger simply returns from the event.

AppleScript also enables creation of sophisticated interface builders. The interface elements post messages when a user interacts with them. The user arranges the elements into windows, menus, and dialogs. Scripts may be attached to any object in the interface to intercept the messages being sent by the interface elements and provide sophisticated behavior and linking between the elements. Early application builders included Frontmost™, a window and dialog

builder, and AgentBuilder™, which specialized in communication front-ends. Version 2.2 of HyperCard, released in 1992, included support for OSA, so that AppleScript or any OSA language could be used in place of HyperTalk.

Two major application builders have emerged recently. FaceSpan, originally released in 1994, has grown into a full-featured application development tool. FaceSpan includes an integrated script debugger. Apple released AppleScript Studio in 2002 as part of its XCode development platform. A complete application can be developed with a wide range of standard user interface elements to which scripts can be attached. AppleScript Studio won Macworld Best of Show Awards at the 2001 Seybold Conference in San Francisco.

In 2005 Apple released Automator, a tool for creating sequences of actions that define workflows. Automator sequences are not stored or executed as AppleScripts, but can contain AppleScripts as primitive actions. The most interesting thing about Automator is that each action has an input and an output, much like a command in a Unix pipe. The resulting model is quite intuitive and easy to use for simple automation tasks.

Although Apple Events are normally handled by applications, it is also possible to install *system event handlers*. When an Apple Event is delivered to an application, the application may handle the event or indicate that it was not handled. When an application does not handle an event, the Apple Event manager searches for a system event handler. System event handlers are packaged in *script extensions* (also known as OSAX) and are installed on the system via Scripting Additions that are loaded when the system starts up.

6.4 Scriptable Applications

Eventually, a wide range of scriptable applications became available: there are currently 160 scriptable applications listed on the Apple web site and the AppleScript sourcebook [8]. Every kind of application is present, including word processors, databases, file compression utilities, and development tools. Many of the most popular applications are scriptable, including Microsoft Office, Adobe Photoshop, Quark Expression, FileMaker, and Illustrator. In addition, most components of the Mac OS are scriptable, including the Finder, QuickTime Player, Address Book, iTunes, Mail, Safari Browser, AppleWorks, DVD Player, Help Viewer, iCal, iChat, iSync, iPhoto, and control panels.

Other systems also benefitted from the infrastructure created by AppleScript. The Macintosh AV™ speech recognition system uses AppleScript, so any scriptable application can be driven using speech.

7. Evolution

After version 1.1, the evolution of AppleScript was driven primarily by changes in the Macintosh OS. Since AppleScript was first released, the operating system has undergone two major shifts, first when Apple moved from the Motorola

68000 to the PowerPC chip, and then when it moved from the Classic Macintosh OS to the Unix-based OS X. Few changes were made to the language itself, while scriptable applications and operating system components experienced rapid expansion and evolution. A detailed history with discussion of new features, bugs, and fixes can be found in the AppleScript Sourcebook [8], which we summarize here.

The first upgrade to AppleScript, version 1.1.2, was created for Macintosh OS 8.0, introduced in July 1997. Despite the rigorous source code configuration process (see Section 5), Apple could not figure out how to compile the system and contracted with Warren Harris to help with the job. A number of bugs were fixed and some small enhancements were made to conform to Macintosh OS 8.0 standards. At the same time several system applications and extensions were changed in ways that could break old scripts. The most important improvement was a new scriptable Finder, which eliminated the need for a Finder scripting extension.

In 1997 AppleScript was at the top of the list of features to eliminate in order to save money. Cal Simone, mentioned in Section 6.3, successfully rallied customers to rescue AppleScript.

In October 1998 Apple released AppleScript 1.3 with UNICODE support recompiled as a native PowerPC extension; however, the Apple Events Manager was still emulated as Motorola 68000 code. The dialect feature was no longer supported; English became the single standard dialect. This version came much closer to realizing the vision of uniform access to all system resources from scripts. At least 30 different system components, including File Sharing, Apple Video Player and Users & Groups, were now scriptable. New scriptable applications appeared as well, including Microsoft Internet Explorer and Outlook Express.

The PowerPC version of AppleScript received an Eddy Award from MacWorld as “Technology of the Year” for 1998 and was also demonstrated in Steve Jobs’ Seybold 1998 address. In 2006, MacWorld placed AppleScript as #17 on its list of the 30 most significant Mac products ever. AppleScript was a long-term investment in fundamental infrastructure that took many years to pay dividends.

The most significant language changes involved the **tell** statement. For example, the machine class used to identify remote applications was extended to accept URLs (see Section 3.2), allowing AppleScript control of remote applications via TCP/IP.

When Mac OS X was released in March 2001, it included AppleScript 1.6. In porting applications and system components to OS X, Apple sometimes sacrificed scripting support. As a result, there was a significant reduction in the number of scriptable applications after the release of OS X. Full scriptability is being restored slowly in later releases.

In October 2006, Google reported an estimated 8,570,000 hits for the word “AppleScript”.

8. Evaluation

AppleScript was developed by a small group with a short schedule, a tight budget and a big job. There was neither time nor money to fully research design choices.

AppleScript and Apple Events introduced a new approach to remote communication in high-latency environments [33]. Object references are symbolic paths, or queries, that identify one or more objects in an application. When a command is applied to an object reference, both the command and the object reference are sent (as an Apple Event containing an object specifier) to the application hosting the target object. The application interprets the object specifier and then performs the action on the specified objects.

In summary, AppleScript views an application as a form of object-oriented database. The application publishes a specialized terminology containing verbs and nouns that describe the logical structure and behavior of its objects. Names in the terminology are composed using a standard query language to create programs that are executed by the remote application. The execution model does not involve remote object references and proxies as in CORBA. Rather than send each field access and method individually to the remote application and creating proxies to represent intermediate values, AppleScript sends the entire command to the remote application for execution. From a pure object-oriented viewpoint, the entire application is the only real object; the “objects” within it are identified only by symbolic references, or queries.

After completing AppleScript, I learned about COM and was impressed with its approach to distributed object-oriented programming. Its consistent use of interfaces enables interoperability between different systems and languages. Although interface negotiation is complex, invoking a method through an interface is highly optimized. This approach allows fine-grained objects that are tightly coupled through shared binary interfaces. For many years I believed that COM and CORBA would beat the AppleScript communication model in the long run. However, recent developments have made me realize that this may not be the case.

AppleScript uses a large-granularity messaging model that has many similarities to the web service standards that began to emerge in 1999 [10]. Both are loosely coupled and support large-granularity communication. Apple Events data descriptors are similar to XML in that they describe arbitrary labeled tree structures without fixed semantics. AppleScript terminologies are similar to web service description language (WSDL) files. It is perhaps not an accident that Dave Winer, who worked extensively with AppleScript and Apple Events, is also one of the original developers of web service models. There may be useful lessons to be learned for web services, given that AppleScript represents a significant body of experience with large-granularity messaging. One difference is that AppleScript includes a standard query model for identifying remote objects. A similar ap-

proach could be useful for web services. As I write in 2006, I suspect that COM and CORBA will be overwhelmed by web services, although the outcome is far from certain now.

AppleScript is also similar to traditional database interfaces like ODBC [38]. In AppleScript the query model is integrated directly into the language, rather than being executed as strings as in ODBC. A similar approach has been adopted by Microsoft for describing queries in .NET languages [3].

User tests revealed that casual users don't easily understand the idea of references, or having multiple references to the same value. It is easier to understand a model in which values are copied or moved, rather than assigning references. The feedback from user tests in early 1993 was too late in the development cycle to address this issue with anything more than a cosmetic change, to use **copy** instead of **set** for assignment.

Writing scriptable applications is difficult. Just as user interface design requires judgment and training, creating a good scripting interface requires a lot of knowledge and careful design. It is too difficult for application developers to create terminologies that work well in the naturalistic grammar. They must pay careful attention to the linguistic properties of the names they choose.

The experiment in designing a language that resembled natural languages (English and Japanese) was not successful. It was assumed that scripts should be presented in "natural language" so that average people could read and write them. This lead to the invention of multi-token keywords and the ability to disambiguate tokens without spaces for Japanese Kanji. In the end the syntactic variations and flexibility did more to confuse programmers than to help them out. It is not clear whether it is easier for novice users to work with a scripting language that resembles natural language, with all its special cases and idiosyncrasies. The main problem is that AppleScript only appears to be a natural language: in fact, it is an artificial language, like any other programming language. Recording was successful, but even small changes to the script may introduce subtle syntactic errors that baffle users. It is easy to read AppleScript, but quite hard to write it.

When writing programs or scripts, users prefer a more conventional programming language structure. Later versions of AppleScript dropped support for dialects. In hindsight, we believe that AppleScript should have adopted the Professional Dialect that was developed but never shipped.

Finally, readability was no substitute for an effective security mechanism. Most people just run scripts—they don't read or write them.

9. Conclusion

AppleScript is widely used today and is a core technology of Mac OS X. Many applications, including Quark Express, Microsoft Office, and FileMaker, support scripting.

Small scripts are used to automate repetitive tasks. Larger scripts have been developed for database publishing, document preparation, and even web applications.

There are many interesting lessons to be learned from AppleScript. On a technical level, its model of pluggable embedded scripting languages has become commonplace. The communication mechanism of Apple Events, which is certainly inferior to RPC mechanisms for single-machine or in-process interactions, may turn out to be a good model for large-granularity communication models such as web services. Many of the current problems in AppleScript can be traced to the use of syntax based on natural language; however, the ability to create pluggable dialects may provide a solution in the future, by creating a new syntax based on conventional programming languages.

Acknowledgments

Thanks to Jens Alfke, Paul Berkowitz, Bill Cheeseman, Chris Espinosa, Michael Farr, Steve Goldband Tom Hammer, David Hoerl, Alexander Kellett, Wayne Malkin, Matt Neuburg, Chuck Piercy, Hamish Sanderson, and Stephen Weyl, for discussions about this paper. Special thanks to Andrew Black and Kathleen Fisher for their guidance, encouragement, flexibility, and careful reading of my work in progress.

References

- [1] Alfred V. Aho and Jeffrey D. Ullman. *Principles of Compiler Design* (Addison-Wesley series in computer science and information processing). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1977.
- [2] Apple Computer Inc. *AppleScript Language Guide*. Addison-Wesley, 1993.
- [3] Gavin Bierman, Erik Meijer, and Wolfram Schulte. The essence of data access in cw. In *European Conference on Object-Oriented Programming*. Springer Verlag, 2005.
- [4] Don Box, David EhneBuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielson, Satish Thatte, and Dave Winer. Simple object access protocol 1.1. <http://www.w3.org/TR/SOAP>.
- [5] Gilad Bracha and William Cook. Mixin-based inheritance. In *Proc. of ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications*, pages 303–311, 1990.
- [6] Peter Canning, William Cook, Walt Hill, John Mitchell, and Walter Olthoff. F-bounded polymorphism for object-oriented programming. In *Proc. of Conf. on Functional Programming Languages and Computer Architecture*, pages 273–280, 1989.
- [7] Peter Canning, William Cook, Walt Hill, and Walter Olthoff. Interfaces for strongly-typed object-oriented programming. In *Proc. of ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications*, pages 457–467, 1989.
- [8] Bill Cheeseman. Applescript sourcebook. <http://www.AppleScriptSourcebook.com>.

- [9] Peter P. Chen. The entity-relationship model — toward a unified view of data. *ACM Transactions on Database Systems (TODS)*, 1(1):9–36, 1976.
- [10] Roberto Chinnici, Martin Gudgin, Jean-Jacques Moreau, and Sanjiva Weerawarana. Web Services Description Language Version 1.2, July 2002. W3C Working Draft 9.
- [11] Apple Computer. *Inside Macintosh: Interrapplication Communication*. Addison-Wesley, 1991.
- [12] Apple Computer. *Inside Macintosh: Macintosh Toolbox Essentials*. Addison-Wesley, 1991.
- [13] Apple Computer. ichat 2.0 dictionary. FooDoo Lounge web site by Richard Morton, 2002-2005.
- [14] William Cook. *A Denotational Semantics of Inheritance*. PhD thesis, Brown University, 1989.
- [15] William Cook. A proposal for making Eiffel type-safe. In *Proc. European Conf. on Object-Oriented Programming*, pages 57–70. British Computing Society Workshop Series, 1989. Also in *The Computer Journal*, 32(4):305–311, 1989.
- [16] William Cook. Object-oriented programming versus abstract data types. In *Proc. of the REX Workshop/School on the Foundations of Object-Oriented Languages*, volume 173 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.
- [17] William Cook. Interfaces and specifications for the Smalltalk collection classes. In *Proc. of ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications*, 1992.
- [18] William Cook, Walt Hill, and Peter Canning. Inheritance is not subtyping. In *Proc. of the ACM Symp. on Principles of Programming Languages*, pages 125–135, 1990.
- [19] William Cook and Jens Palsberg. A denotational semantics of inheritance and its correctness. In *Proc. of ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications*, pages 433–444, 1989.
- [20] William R. Cook and Victor Law. An algorithm editor for structured design (abstract). In *Proc. of the ACM Computer Science Conference*, 1983.
- [21] Allen Cypher. Eager: programming repetitive tasks by example. In *CHI '91: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 33–39, New York, NY, USA, 1991. ACM Press.
- [22] Allen Cypher, editor. *Watch What I Do – Programming by Demonstration*. MIT Press, Cambridge, MA, USA, 1993. Full text available at web.media.mit.edu/~lieber/PBE/.
- [23] A. Goldberg and D. Robson. *Smalltalk-80: the Language and Its Implementation*. Addison-Wesley, 1983.
- [24] A. Goldstein. *AppleScript: The Missing Manual*. O'Reilly, 2005.
- [25] Warren Harris. Abel posthumous report. HP Labs, 1993.
- [26] Apple Computer Inc. Scripting interface guidelines. Technical Report TN2106, Apple Computer Inc.
- [27] Apple Computer Inc. *HyperCard User's Guide*. Addison Wesley, 1987.
- [28] Apple Computer Inc. *HyperCard Script Language Guide*: *The HyperTalk Language*. Addison Wesley, 1988.
- [29] Steven C. Johnson. Yacc: Yet another compiler compiler. In *UNIX Programmer's Manual*, volume 2, pages 353–387. Holt, Rinehart, and Winston, New York, NY, USA, 1979.
- [30] S. Michel. *HyperCard: The Complete Reference*. Osborne McGrawHill, 1989.
- [31] M. Neuburg. *AppleScript : The Definitive Guide*. O'Reilly, 2003.
- [32] Object Management Group. *OMG Unified Modeling Language Specification Version 1.5*, March 2003.
- [33] David A. Patterson. Latency lags bandwidth. *Commun. ACM*, 47(10):71–75, 2004.
- [34] Trygve Reenskaug. Models — views — controllers. Technical report, Xerox PARC, December 1979.
- [35] Alan Snyder. The essence of objects: Concepts and terms. *IEEE Softw.*, 10(1):31–42, 1993.
- [36] Lynn Andrea Stein, Henry Lieberman, and David Ungar. A shared view of sharing: The treaty of orlando. In Won Kim and Frederick H. Lochovsky, editors, *Object-Oriented Concepts, Databases and Applications*, pages 31–48. ACM Press and Addison-Wesley, 1989.
- [37] D. Ungar and R.B. Smith. Self: The power of simplicity. In *Proc. of ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications*, pages 227–242, 1987.
- [38] Murali Venkatrao and Michael Pizzo. SQL/CLI – a new binding style for SQL. *SIGMOD Record*, 24(4):72–77, 1995.
- [39] Steven R. Wood. Z — the 95% program editor. In *Proceedings of the ACM SIGPLAN SIGOA symposium on Text manipulation*, pages 1–7, New York, NY, USA, 1981. ACM Press.

The Evolution of Lua

Roberto Ierusalimschy

Department of Computer Science,
PUC-Rio, Rio de Janeiro, Brazil
roberto@inf.puc-rio.br

Luiz Henrique de Figueiredo

IMPA–Instituto Nacional de
Matemática Pura e Aplicada, Brazil
lhf@impa.br

Waldemar Celes

Department of Computer Science,
PUC-Rio, Rio de Janeiro, Brazil
celes@inf.puc-rio.br

Abstract

We report on the birth and evolution of Lua and discuss how it moved from a simple configuration language to a versatile, widely used language that supports extensible semantics, anonymous functions, full lexical scoping, proper tail calls, and coroutines.

Categories and Subject Descriptors K.2 [HISTORY OF COMPUTING]: Software; D.3 [PROGRAMMING LANGUAGES]

1. Introduction

Lua is a scripting language born in 1993 at PUC-Rio, the Pontifical Catholic University of Rio de Janeiro in Brazil. Since then, Lua has evolved to become widely used in all kinds of industrial applications, such as robotics, literate programming, distributed business, image processing, extensible text editors, Ethernet switches, bioinformatics, finite-element packages, web development, and more [2]. In particular, Lua is one of the leading scripting languages in game development.

Lua has gone far beyond our most optimistic expectations. Indeed, while almost all programming languages come from North America and Western Europe (with the notable exception of Ruby, from Japan) [4], Lua is the only language created in a developing country to have achieved global relevance.

From the start, Lua was designed to be simple, small, portable, fast, and easily embedded into applications. These design principles are still in force, and we believe that they account for Lua's success in industry. The main characteristic of Lua, and a vivid expression of its simplicity, is that it offers a single kind of data structure, the *table*, which is the Lua term for an associative array [9]. Although most script-

Permission to make digital/hard copy of part of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date of appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Permission may be requested from the Publications Dept., ACM, Inc., 2 Penn Plaza, New York, NY 11201-0701, USA, fax:+1(212) 869-0481, permissions@acm.org

©2007 ACM 978-1-59593-766-7/2007/06-ART2 \$5.00
DOI 10.1145/1238844.1238846

<http://doi.acm.org/10.1145/1238844.1238846>

ing languages offer associative arrays, in no other language do associative arrays play such a central role. Lua tables provide simple and efficient implementations for modules, prototype-based objects, class-based objects, records, arrays, sets, bags, lists, and many other data structures [28].

In this paper, we report on the birth and evolution of Lua. We discuss how Lua moved from a simple configuration language to a powerful (but still simple) language that supports extensible semantics, anonymous functions, full lexical scoping, proper tail calls, and coroutines. In §2 we give an overview of the main concepts in Lua, which we use in the other sections to discuss how Lua has evolved. In §3 we relate the prehistory of Lua, that is, the setting that led to its creation. In §4 we relate how Lua was born, what its original design goals were, and what features its first version had. A discussion of how and why Lua has evolved is given in §5. A detailed discussion of the evolution of selected features is given in §6. The paper ends in §7 with a retrospective of the evolution of Lua and in §8 with a brief discussion of the reasons for Lua's success, especially in games.

2. Overview

In this section we give a brief overview of the Lua language and introduce the concepts discussed in §5 and §6. For a complete definition of Lua, see its reference manual [32]. For a detailed introduction to Lua, see Roberto's book [28]. For concreteness, we shall describe Lua 5.1, which is the current version at the time of this writing (April 2007), but most of this section applies unchanged to previous versions.

Syntactically, Lua is reminiscent of Modula and uses familiar keywords. To give a taste of Lua's syntax, the code below shows two implementations of the factorial function, one recursive and another iterative. Anyone with a basic knowledge of programming can probably understand these examples without explanation.

```
function factorial(n)          function factorial(n)
    if n == 0 then            local a = 1
        return 1              for i = 1,n do
    else                      a = a*i
        return n*factorial(n-1) end
    end                        return a
end
```

Semantically, Lua has many similarities with Scheme, even though these similarities are not immediately clear because the two languages are syntactically very different. The influence of Scheme on Lua has gradually increased during Lua's evolution: initially, Scheme was just a language in the background, but later it became increasingly important as a source of inspiration, especially with the introduction of anonymous functions and full lexical scoping.

Like Scheme, Lua is dynamically typed: variables do not have types; only values have types. As in Scheme, a variable in Lua never contains a structured value, only a reference to one. As in Scheme, a function name has no special status in Lua: it is just a regular variable that happens to refer to a function value. Actually, the syntax for function definition ‘`function foo() ... end`’ used above is just syntactic sugar for the assignment of an anonymous function to a variable: ‘`foo = function () ... end`’. Like Scheme, Lua has first-class functions with lexical scoping. Actually, all values in Lua are first-class values: they can be assigned to global and local variables, stored in tables, passed as arguments to functions, and returned from functions.

One important semantic difference between Lua and Scheme—and probably the main distinguishing feature of Lua—is that Lua offers *tables* as its sole data-structuring mechanism. Lua tables are associative arrays [9], but with some important features. Like all values in Lua, tables are first-class values: they are not bound to specific variable names, as they are in Awk and Perl. A table can have any value as key and can store any value. Tables allow simple and efficient implementation of records (by using field names as keys), sets (by using set elements as keys), generic linked structures, and many other data structures. Moreover, we can use a table to implement an array by using natural numbers as indices. A careful implementation [31] ensures that such a table uses the same amount of memory that an array would (because it is represented internally as an actual array) and performs better than arrays in similar languages, as independent benchmarks show [1].

Lua offers an expressive syntax for creating tables in the form of *constructors*. The simplest constructor is the expression ‘`{}`’, which creates a new, empty table. There are also constructors to create lists (or arrays), such as

```
{ "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" }
```

and to create records, such as

```
{lat= -22.90, long= -43.23, city= "Rio de Janeiro"}
```

These two forms can be freely mixed. Tables are indexed using square brackets, as in ‘`t[2]`’, with ‘`t.x`’ as sugar for ‘`t["x"]`’.

The combination of table constructors and functions turns Lua into a powerful general-purpose procedural data-description language. For instance, a bibliographic database in a format similar to the one used in BibTeX [34] can be written as a series of table constructors such as this:

```
article{"spe96",
  authors = {"Roberto Ierusalimschy",
             "Luiz Henrique de Figueiredo",
             "Waldemar Celes"},
  title = "Lua: an Extensible Extension Language",
  journal = "Software: Practice & Experience",
  year = 1996,
}
```

Although such a database seems to be an inert data file, it is actually a valid Lua program: when the database is loaded into Lua, each item in it invokes a function, because ‘`article{...}`’ is syntactic sugar for ‘`article({...})`’, that is, a function call with a table as its single argument. It is in this sense that such files are called procedural data files.

We say that Lua is an extensible extension language [30]. It is an *extension language* because it helps to extend applications through configuration, macros, and other end-user customizations. Lua is designed to be embedded into a host application so that users can control how the application behaves by writing Lua programs that access application services and manipulate application data. It is *extensible* because it offers *userdata* values to hold application data and *extensible semantics* mechanisms to manipulate these values in natural ways. Lua is provided as a small core that can be extended with user functions written in both Lua and C. In particular, input and output, string manipulation, mathematical functions, and interfaces to the operating system are all provided as external libraries.

Other distinguishing features of Lua come from its implementation:

Portability: Lua is easy to build because it is implemented in strict ANSI C.¹ It compiles out-of-the-box on most platforms (Linux, Unix, Windows, Mac OS X, etc.), and runs with at most a few small adjustments in virtually all platforms we know of, including mobile devices (e.g., handheld computers and cell phones) and embedded microprocessors (e.g., ARM and Rabbit). To ensure portability, we strive for warning-free compilations under as many compilers as possible.

Ease of embedding: Lua has been designed to be easily embedded into applications. An important part of Lua is a well-defined application programming interface (API) that allows full communication between Lua code and external code. In particular, it is easy to extend Lua by exporting C functions from the host application. The API allows Lua to interface not only with C and C++, but also with other languages, such as Fortran, Java, Smalltalk, Ada, C# (.Net), and even with other scripting languages (e.g., Perl and Ruby).

¹ Actually, Lua is implemented in “clean C”, that is, the intersection of C and C++. Lua compiles unmodified as a C++ library.

Small size: Adding Lua to an application does not bloat it.

The whole Lua distribution, including source code, documentation, and binaries for some platforms, has always fit comfortably on a floppy disk. The tarball for Lua 5.1, which contains source code, documentation, and examples, takes 208K compressed and 835K uncompressed. The source contains around 17,000 lines of C. Under Linux, the Lua interpreter built with all standard Lua libraries takes 143K. The corresponding numbers for most other scripting languages are more than an order of magnitude larger, partially because Lua is primarily meant to be embedded into applications and so its official distribution includes only a few libraries. Other scripting languages are meant to be used standalone and include many libraries.

Efficiency: Independent benchmarks [1] show Lua to be one of the fastest languages in the realm of interpreted scripting languages. This allows application developers to write a substantial fraction of the whole application in Lua. For instance, over 40% of Adobe Lightroom is written in Lua (that represents around 100,000 lines of Lua code).

Although these are features of a specific implementation, they are possible only due to the design of Lua. In particular, Lua's simplicity is a key factor in allowing a small, efficient implementation [31].

3. Prehistory

Lua was born in 1993 inside Tecgraf, the Computer Graphics Technology Group of PUC-Rio in Brazil. The creators of Lua were Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. Roberto was an assistant professor at the Department of Computer Science of PUC-Rio. Luiz Henrique was a post-doctoral fellow, first at IMPA and later at Tecgraf. Waldemar was a Ph.D. student in Computer Science at PUC-Rio. All three were members of Tecgraf, working on different projects there before getting together to work on Lua. They had different, but related, backgrounds: Roberto was a computer scientist interested mainly in programming languages; Luiz Henrique was a mathematician interested in software tools and computer graphics; Waldemar was an engineer interested in applications of computer graphics. (In 2001, Waldemar joined Roberto as faculty at PUC-Rio and Luiz Henrique became a researcher at IMPA.)

Tecgraf is a large research and development laboratory with several industrial partners. During the first ten years after its creation in May 1987, Tecgraf focused mainly on building basic software tools to enable it to produce the interactive graphical programs needed by its clients. Accordingly, the first Tecgraf products were drivers for graphical terminals, plotters, and printers; graphical libraries; and graphical interface toolkits. From 1977 until 1992, Brazil had a pol-

icy of strong trade barriers (called a “market reserve”) for computer hardware and software motivated by a nationalistic feeling that Brazil could and should produce its own hardware and software. In that atmosphere, Tecgraf’s clients could not afford, either politically or financially, to buy customized software from abroad: by the market reserve rules, they would have to go through a complicated bureaucratic process to prove that their needs could not be met by Brazilian companies. Added to the natural geographical isolation of Brazil from other research and development centers, those reasons led Tecgraf to implement from scratch the basic tools it needed.

One of Tecgraf’s largest partners was (and still is) Petrobras, the Brazilian oil company. Several Tecgraf products were interactive graphical programs for engineering applications at Petrobras. By 1993, Tecgraf had developed little languages for two of those applications: a data-entry application and a configurable report generator for lithology profiles. These languages, called DEL and SOL, were the ancestors of Lua. We describe them briefly here to show where Lua came from.

3.1 DEL

The engineers at Petrobras needed to prepare input data files for numerical simulators several times a day. This process was boring and error-prone because the simulation programs were legacy code that needed strictly formatted input files—typically bare columns of numbers, with no indication of what each number meant, a format inherited from the days of punched cards. In early 1992, Petrobras asked Tecgraf to create at least a dozen graphical front-ends for this kind of data entry. The numbers would be input interactively, just by clicking on the relevant parts of a diagram describing the simulation—a much easier and more meaningful task for the engineers than editing columns of numbers. The data file, in the correct format for the simulator, would be generated automatically. Besides simplifying the creation of data files, such front-ends provided the opportunity to add data validation and also to compute derived quantities from the input data, thus reducing the amount of data needed from the user and increasing the reliability of the whole process.

To simplify the development of those front-ends, a team led by Luiz Henrique de Figueiredo and Luiz Cristovão Gomes Coelho decided to code all front-ends in a uniform way, and so designed DEL (“data-entry language”), a simple declarative language to describe each data-entry task [17]. DEL was what is now called a domain-specific language [43], but was then simply called a little language [10].

A typical DEL program defined several “entities”. Each entity could have several fields, which were named and typed. For implementing data validation, DEL had predicate statements that imposed restrictions on the values of entities. DEL also included statements to specify how data was to be input and output. An entity in DEL was essentially what is called a structure or record in conventional

programming languages. The important difference—and what made DEL suitable for the data-entry problem—is that entity names also appeared in a separate graphics metafile, which contained the associated diagram over which the engineer did the data entry. A single interactive graphical interpreter called ED (an acronym for ‘entrada de dados’, which means ‘data entry’ in Portuguese) was written to interpret DEL programs. All those data-entry front-ends requested by Petrobras were implemented as DEL programs that ran under this single graphical application.

DEL was a success both among the developers at Tecgraf and among the users at Petrobras. At Tecgraf, DEL simplified the development of those front-ends, as originally intended. At Petrobras, DEL allowed users to tailor data-entry applications to their needs. Soon users began to demand more power from DEL, such as boolean expressions for controlling whether an entity was active for input or not, and DEL became heavier. When users began to ask for control flow, with conditionals and loops, it was clear that ED needed a real programming language instead of DEL.

3.2 SOL

At about the same time that DEL was created, a team lead by Roberto Ierusalimschy and Waldemar Celes started working on PGM, a configurable report generator for lithology profiles, also for Petrobras. The reports generated by PGM consisted of several columns (called “tracks”) and were highly configurable: users could create and position the tracks, and could choose colors, fonts, and labels; each track could have a grid, which also had its set of options (log/linear, vertical and horizontal ticks, etc.); each curve had its own scale, which had to be changed automatically in case of overflow; etc. All this configuration was to be done by the end-users, typically geologists and engineers from Petrobras working in oil plants and off-shore platforms. The configurations had to be stored in files, for reuse. The team decided that the best way to configure PGM was through a specialized description language called SOL, an acronym for *Simple Object Language*.

Because PGM had to deal with many different objects, each with many different attributes, the SOL team decided not to fix those objects and attributes into the language. Instead, SOL allowed type declarations, as in the code below:

```
type @track{ x:number, y:number=23, id=0 }
type @line{ t:@track{x=8}, z:number* }
T = @track{ y=9, x=10, id="1992-34" }
L = @line{ t=@track{x=T.y, y=T.x}, z=[2,3,4] }
```

This code defines two types, `track` and `line`, and creates two objects, a `track` `T` and a `line` `L`. The `track` type contains two numeric attributes, `x` and `y`, and an untyped attribute, `id`; attributes `y` and `id` have default values. The `line` type contains a `track` `t` and a list of numbers `z`. The `track` `t` has as default value a `track` with `x=8`, `y=23`, and `id=0`. The syntax

of SOL was strongly influenced by BibTeX [34] and UIL, a language for describing user interfaces in Motif [39].

The main task of the SOL interpreter was to read a report description, check whether the given objects and attributes were correctly typed, and then present the information to the main program (PGM). To allow the communication between the main program and the SOL interpreter, the latter was implemented as a C library that was linked to the main program. The main program could access all configuration information through an API in this library. In particular, the main program could register a callback function for each type, which the SOL interpreter would call to create an object of that type.

4. Birth

The SOL team finished an initial implementation of SOL in March 1993, but they never delivered it. PGM would soon require support for procedural programming to allow the creation of more sophisticated layouts, and SOL would have to be extended. At the same time, as mentioned before, ED users had requested more power from DEL. ED also needed further descriptive facilities for programming its user interface. Around mid-1993, Roberto, Luiz Henrique, and Waldemar got together to discuss DEL and SOL, and concluded that the two languages could be replaced by a single, more powerful language, which they decided to design and implement. Thus the Lua team was born; it has not changed since.

Given the requirements of ED and PGM, we decided that we needed a real programming language, with assignments, control structures, subroutines, etc. The language should also offer data-description facilities, such as those offered by SOL. Moreover, because many potential users of the language were not professional programmers, the language should avoid cryptic syntax and semantics. The implementation of the new language should be highly portable, because Tecgraf’s clients had a very diverse collection of computer platforms. Finally, since we expected that other Tecgraf products would also need to embed a scripting language, the new language should follow the example of SOL and be provided as a library with a C API.

At that point, we could have adopted an existing scripting language instead of creating a new one. In 1993, the only real contender was Tcl [40], which had been explicitly designed to be embedded into applications. However, Tcl had unfamiliar syntax, did not offer good support for data description, and ran only on Unix platforms. We did not consider LISP or Scheme because of their unfriendly syntax. Python was still in its infancy. In the free, do-it-yourself atmosphere that then reigned in Tecgraf, it was quite natural that we should try to develop our own scripting language. So, we started working on a new language that we hoped would be simpler to use than existing languages. Our original design decisions were: keep the language simple and small, and keep the im-

plementation simple and portable. Because the new language was partially inspired by SOL (*sun* in Portuguese), a friend at Tecgraf (Carlos Henrique Levy) suggested the name ‘Lua’ (*moon* in Portuguese), and Lua was born. (DEL did not influence Lua as a language. The main influence of DEL on the birth of Lua was rather the realization that large parts of complex applications could be written using embeddable scripting languages.)

We wanted a light full language with data-description facilities. So we took SOL’s syntax for record and list construction (but not type declaration), and unified their implementation using tables: records use strings (the field names) as indices; lists use natural numbers. An assignment such as

```
T = @track{ y=9, x=10, id="1992-34" }
```

which was valid in SOL, remained valid in Lua, but with a different meaning: it created an object (that is, a table) with the given fields, and then called the function `track` on this table to validate the object or perhaps to provide default values to some of its fields. The final value of the expression was that table.

Except for its procedural data-description constructs, Lua introduced no new concepts: Lua was created for production use, not as an academic language designed to support research in programming languages. So, we simply borrowed (even unconsciously) things that we had seen or read about in other languages. We did not reread old papers to remember details of existing languages. We just started from what we knew about other languages and reshaped that according to our tastes and needs.

We quickly settled on a small set of control structures, with syntax mostly borrowed from Modula (`while`, `if`, and `repeat until`). From CLU we took multiple assignment and multiple returns from function calls. We regarded multiple returns as a simpler alternative to reference parameters used in Pascal and Modula and to in-out parameters used in Ada; we also wanted to avoid explicit pointers (used in C). From C++ we took the neat idea of allowing a local variable to be declared only where we need it. From SNOBOL and Awk we took associative arrays, which we called tables; however, tables were to be objects in Lua, not attached to variables as in Awk.

One of the few (and rather minor) innovations in Lua was the syntax for string concatenation. The natural ‘+’ operator would be ambiguous, because we wanted automatic coercion of strings to numbers in arithmetic operations. So, we invented the syntax ‘..’ (two dots) for string concatenation.

A polemic point was the use of semicolons. We thought that requiring semicolons could be a little confusing for engineers with a Fortran background, but not allowing them could confuse those with a C or Pascal background. In typical committee fashion, we settled on optional semicolons.

Initially, Lua had seven types: numbers (implemented solely as reals), strings, tables, nil, userdata (pointers to C objects), Lua functions, and C functions. To keep the language small, we did not initially include a boolean type: as in Lisp, nil represented *false* and any other value represented *true*. Over 13 years of continuous evolution, the only changes in Lua types were the unification of Lua functions and C functions into a single function type in Lua 3.0 (1997) and the introduction of booleans and threads in Lua 5.0 (2003) (see §6.1). For simplicity, we chose to use dynamic typing instead of static typing. For applications that needed type checking, we provided basic reflective facilities, such as run-time type information and traversal of the global environment, as built-in functions (see §6.11).

By July 1993, Waldemar had finished the first implementation of Lua as a course project supervised by Roberto. The implementation followed a tenet that is now central to Extreme Programming: “the simplest thing that could possibly work” [7]. The lexical scanner was written with `lex` and the parser with `yacc`, the classic Unix tools for implementing languages. The parser translated Lua programs into instructions for a stack-based virtual machine, which were then executed by a simple interpreter. The C API made it easy to add new functions to Lua, and so this first version provided only a tiny library of five built-in functions (`next`, `nextvar`, `print`, `tonumber`, `type`) and three small external libraries (`input` and `output`, mathematical functions, and string manipulation).

Despite this simple implementation—or possibly because of it—Lua surpassed our expectations. Both PGM and ED used Lua successfully (PGM is still in use today; ED was replaced by EDG [12], which was mostly written in Lua). Lua was an immediate success in Tecgraf and soon other projects started using it. This initial use of Lua at Tecgraf was reported in a brief talk at the VII Brazilian Symposium on Software Engineering, in October 1993 [29].

The remainder of this paper relates our journey in improving Lua.

5. History

Figure 1 shows a timeline of the releases of Lua. As can be seen, the time interval between versions has been gradually increasing since Lua 3.0. This reflects our perception that

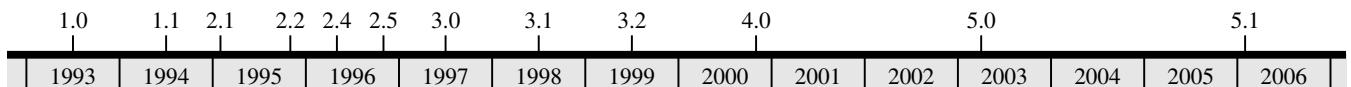


Figure 1. The releases of Lua.

	1.0	1.1	2.1	2.2	2.4	2.5	3.0	3.1	3.2	4.0	5.0	5.1
constructors	●	●	●	●	●	●	●	●	●	●	●	●
garbage collection	●	●	●	●	●	●	●	●	●	●	●	●
extensible semantics	○	○	●	●	●	●	●	●	●	●	●	●
support for OOP	○	○	●	●	●	●	●	●	●	●	●	●
long strings	○	○	○	●	●	●	●	●	●	●	●	●
debug API	○	○	○	●	●	●	●	●	●	●	●	●
external compiler	○	○	○	○	●	●	●	●	●	●	●	●
vararg functions	○	○	○	○	○	●	●	●	●	●	●	●
pattern matching	○	○	○	○	○	●	●	●	●	●	●	●
conditional compilation	○	○	○	○	○	○	●	●	●	○	○	○
anonymous functions, closures	○	○	○	○	○	○	○	●	●	●	●	●
debug library	○	○	○	○	○	○	○	○	●	●	●	●
multi-state API	○	○	○	○	○	○	○	○	○	●	●	●
for statement	○	○	○	○	○	○	○	○	○	●	●	●
long comments	○	○	○	○	○	○	○	○	○	○	●	●
full lexical scoping	○	○	○	○	○	○	○	○	○	○	●	●
booleans	○	○	○	○	○	○	○	○	○	○	●	●
coroutines	○	○	○	○	○	○	○	○	○	○	●	●
incremental garbage collection	○	○	○	○	○	○	○	○	○	○	○	●
module system	○	○	○	○	○	○	○	○	○	○	○	●
	1.0	1.1	2.1	2.2	2.4	2.5	3.0	3.1	3.2	4.0	5.0	5.1
libraries	4	4	4	4	4	4	4	5	6	8	9	
built-in functions	5	7	11	11	13	14	25	27	35	0	0	0
API functions	30	30	30	30	32	32	33	47	41	60	76	79
vm type (stack × register)	S	S	S	S	S	S	S	S	S	R	R	
vm instructions	64	65	69	67	67	68	69	128	64	49	35	38
keywords	16	16	16	16	16	16	16	16	16	18	21	21
other tokens	21	21	23	23	23	23	24	25	25	25	24	26

Table 1. The evolution of features in Lua.

Lua was becoming a mature product and needed stability for the benefit of its growing community. Nevertheless, the need for stability has not hindered progress. Major new versions of Lua, such as Lua 4.0 and Lua 5.0, have been released since then.

The long times between versions also reflects our release model. Unlike other open-source projects, our alpha versions are quite stable and beta versions are essentially final, except for uncovered bugs.² This release model has proved to be good for Lua stability. Several products have been shipped with alpha or beta versions of Lua and worked fine. However, this release model did not give users much chance to experiment with new versions; it also deprived us of timely feedback on proposed changes. So, during the development of Lua 5.0 we started to release “work” versions, which are just snapshots of the current development of Lua. This move brought our current release model closer to the “Release Early, Release Often” motto of the open-source community.

²The number of bugs found after final versions were released has been consistently small: only 10 in Lua 4.0, 17 in Lua 5.0, and 10 in Lua 5.1 so far, none of them critical bugs.

In the remainder of this section we discuss some milestones in the evolution of Lua. Details on the evolution of several specific features are given in §6. Table 1 summarizes this evolution. It also contains statistics about the size of Lua, which we now discuss briefly.

The number of standard libraries has been kept small because we expect that most Lua functions will be provided by the host application or by third-party libraries. Until Lua 3.1, the only standard libraries were for input and output, string manipulation, mathematical functions, and a special library of built-in functions, which did not use the C API but directly accessed the internal data structures. Since then, we have added libraries for debugging (Lua 3.2), interfacing with the operating system (Lua 4.0), tables and coroutines (Lua 5.0), and modules (Lua 5.1).

The size of C API changed significantly when it was redesigned in Lua 4.0. Since then, it has moved slowly toward completeness. As a consequence, there are no longer any built-in functions: all standard libraries are implemented on top of the C API, without accessing the internals of Lua.

The virtual machine, which executes Lua programs, was stack-based until Lua 4.0. In Lua 3.1 we added variants

for many instructions, to try to improve performance. However, this turned out to be too complicated for little performance gain and we removed those variants in Lua 3.2. Since Lua 5.0, the virtual machine is register-based [31]. This change gave the code generator more opportunities for optimization and reduced the number of instructions of typical Lua programs. (Instruction dispatch is a significant fraction of the time spent in the virtual machine [13].) As far as we know, the virtual machine of Lua 5.0 was the first register-based virtual machine to have wide use.

5.1 Lua 1

The initial implementation of Lua was a success in Tecgraf and Lua attracted users from other Tecgraf projects. New users create new demands. Several users wanted to use Lua as the support language for graphics metafiles, which abounded in Tecgraf. Compared with other programmable metafiles, Lua metafiles have the advantage of being based on a truly procedural language: it is natural to model complex objects by combining procedural code fragments with declarative statements. In contrast, for instance, VRML [8] must use another language (Javascript) to model procedural objects.

The use of Lua for this kind of data description, especially large graphics metafiles, posed challenges that were unusual for typical scripting languages. For instance, it was not uncommon for a diagram used in the data-entry program ED to have several thousand parts described by a single Lua table constructor with several thousand items. That meant that Lua had to cope with huge programs and huge expressions. Because Lua precompiled all programs to bytecode for a virtual machine on the fly, it also meant that the Lua compiler had to run fast, even for large programs.

By replacing the lex-generated scanner used in the first version by a hand-written one, we almost doubled the speed of the Lua compiler on typical metafiles. We also modified Lua's virtual machine to handle a long constructor by adding key-value pairs to the table in batches, not individually as in the original virtual machine. These changes solved the initial demands for better performance. Since then, we have always tried to reduce the time spent on precompilation.

In July 1994, we released a new version of Lua with those optimizations. This release coincided with the publication of the first paper describing Lua, its design, and its implementation [15]. We named the new version ‘Lua 1.1’. The previous version, which was never publicly released, was then named ‘Lua 1.0’. (A snapshot of Lua 1.0 taken in July 1993 was released in October 2003 to celebrate 10 years of Lua.)

Lua 1.1 was publicly released as software available in source code by ftp, before the open-source movement got its current momentum. Lua 1.1 had a restrictive user license: it was freely available for academic purposes but commercial uses had to be negotiated. That part of the license did not work: although we had a few initial contacts, no commercial uses were ever negotiated. This and the fact that other

scripting languages (e.g., Tcl) were free made us realize that restrictions on commercial uses might even discourage academic uses, since some academic projects plan to go to market eventually. So, when the time came to release the next version (Lua 2.1), we chose to release it as unrestricted free software. Naively, we wrote our own license text as a slight collage and rewording of existing licenses. We thought it was clear that the new license was quite liberal. Later, however, with the spread of open-source licenses, our license text became a source of noise among some users; in particular, it was not clear whether our license was compatible with GPL. In May 2002, after a long discussion in the mailing list, we decided to release future versions of Lua (starting with Lua 5.0) under the well-known and very liberal MIT license [3]. In July 2002, the Free Software Foundation confirmed that our previous license was compatible with GPL, but we were already committed to adopting the MIT license. Questions about our license have all but vanished since then.

5.2 Lua 2

Despite all the hype surrounding object-oriented programming (which in the early 1990s had reached its peak) and the consequent user pressure to add object-oriented features to Lua, we did not want to turn Lua into an object-oriented language because we did not want to fix a programming paradigm for Lua. In particular, we did not think that Lua needed objects and classes as primitive language concepts, especially because they could be implemented with tables if needed (a table can hold both object data and methods, since functions are first-class values). Despite recurring user pressure, we have not changed our minds to this day: Lua does not force any object or class model onto the programmer. Several object models have been proposed and implemented by users; it is a frequent topic of discussion in our mailing list. We think this is healthy.

On the other hand, we wanted to *allow* object-oriented programming with Lua. Instead of fixing a model, we decided to provide flexible mechanisms that would allow the programmer to build whatever model was suitable to the application. Lua 2.1, released in February 1995, marked the introduction of these *extensible semantics* mechanisms, which have greatly increased the expressiveness of Lua. Extensible semantics has become a hallmark of Lua.

One of the goals of extensible semantics was to allow tables to be used as a basis for objects and classes. For that, we needed to implement inheritance for tables. Another goal was to turn userdata into natural proxies for application data, not merely handles meant to be used solely as arguments to functions. We wanted to be able to index userdata as if they were tables and to call methods on them. This would allow Lua to fulfill one of its main design goals more naturally: to extend applications by providing scriptable access to application services and data. Instead of adding mechanisms to support all these features directly in the language, we decided that it would be conceptually simpler to define a more

general *fallback* mechanism to let the programmer intervene whenever Lua did not know how to proceed.

We introduced fallbacks in Lua 2.1 and defined them for the following operations: table indexing, arithmetic operations, string concatenation, order comparisons, and function calls.³ When one of these operations was applied to the “wrong” kind of values, the corresponding fallback was called, allowing the programmer to determine how Lua would proceed. The table indexing fallbacks allowed userdata (and other values) to behave as tables, which was one of our motivations. We also defined a fallback to be called when a key was absent from a table, so that we could support many forms of inheritance (through delegation). To complete the support for object-oriented programming, we added two pieces of syntactic sugar: method definitions of the form ‘function a:foo(…)' as sugar for ‘function a.foo(self, …)' and method calls of the form ‘a:foo(…)' as sugar for ‘a.foo(a, …)'. In §6.8 we discuss fallbacks in detail and how they evolved into their later incarnations: tag methods and metamethods.

Since Lua 1.0, we have provided introspective functions for values: `type`, which queries the type of a Lua value; `next`, which traverses a table; and `nextvar`, which traverses the global environment. (As mentioned in §4, this was partially motivated by the need to implement SOL-like type checking.) In response to user pressure for full debug facilities, Lua 2.2 (November 1995) introduced a debug API to provide information about running functions. This API gave users the means to write in C their own introspective tools, such as debuggers and profilers. The debug API was initially quite simple: it allowed access to the Lua call stack, to the currently executing line, and provided a function to find the name of a variable holding a given value. Following the M.Sc. work of Tomás Gorham [22], the debug API was improved in Lua 2.4 (May 1996) by functions to access local variables and hooks to be called at line changes and function calls.

With the widespread use of Lua at Tecgraf, many large graphics metafiles were being written in Lua as the output of graphical editors. Loading such metafiles was taking increasingly longer as they became larger and more complex.⁴ Since its first version, Lua precompiled all programs to bytecode just before running them. The load time of a large program could be substantially reduced by saving this bytecode to a file. This would be especially relevant for procedural data files such as graphics metafiles. So, in Lua 2.4, we introduced an external compiler, called `luac`, which precompiled a Lua program and saved the generated bytecode to a binary file. (Our first paper about Lua [15] had already an-

ticipated the possibility of an external compiler.) The format of this file was chosen to be easily loaded and reasonably portable. With `luac`, programmers could avoid parsing and code generation at run time, which in the early days were costly. Besides faster loading, `luac` also allowed off-line syntax checking and protection from casual user changes. Many products (e.g., The Sims and Adobe Lightroom) distribute Lua scripts in precompiled form.

During the implementation of `luac`, we started to restructure Lua’s core into clearly separated modules. As a consequence, it is now quite easy to remove the parsing modules (lexer, parser, and code generator), which currently represent 35% of the core code, leaving just the module that loads precompiled Lua programs, which is merely 3% of the core code. This reduction can be significant when embedding Lua in small devices such as mobile devices, robots and sensors.⁵

Since its first version, Lua has included a library for string-processing. The facilities provided by this library were minimal until Lua 2.4. However, as Lua matured, it became desirable to do heavier text processing in Lua. We thought that a natural addition to Lua would be pattern matching, in the tradition of Snobol, Icon, Awk, and Perl. However, we did not want to include a third-party pattern-matching engine in Lua because such engines tend to be very large; we also wanted to avoid copyright issues that could be raised by including third-party code in Lua.

As a student project supervised by Roberto in the second semester of 1995, Milton Jonathan, Pedro Miller Rabinovitch, Pedro Willemsens, and Vinicius Almendra produced a pattern-matching library for Lua. Experience with that design led us to write our own pattern-matching engine for Lua, which we added to Lua 2.5 (November 1996) in two functions: `strfind` (which originally only found plain substrings) and the new `gsub` function (a name taken from Awk). The `gsub` function globally replaced substrings matching a given pattern in a larger string. It accepted either a replacement string or a function that was called each time a match was found and was intended to return the replacement string for that match. (That was an innovation at the time.) Aiming at a small implementation, we did not include full regular expressions. Instead, the patterns understood by our engine were based on character classes, repetitions, and captures (but not alternation or grouping). Despite its simplicity, this kind of pattern matching is quite powerful and was an important addition to Lua.

That year was a turning point in the history of Lua because it gained international exposure. In June 1996 we published a paper about Lua in *Software: Practice & Experience* [30] that brought external attention to Lua, at least in

³ We also introduced fallbacks for handling fatal errors and for monitoring garbage collection, even though they were not part of extensible semantics.

⁴ Surprisingly, a substantial fraction of the load time was taken in the lexer for converting real numbers from text form to floating-point representation. Real numbers abound in graphics metafiles.

⁵ Crazy Ivan, a robot that won RoboCup in 2000 and 2001 in Denmark, had a “brain” implemented in Lua. It ran directly on a Motorola Coldfire 5206e processor without any operating system (in other words, Lua was the operating system). Lua was stored on a system ROM and loaded programs at startup from the serial port.

academic circles.⁶ In December 1996, shortly after Lua 2.5 was released, the magazine *Dr. Dobb's Journal* featured an article about Lua [16]. *Dr. Dobb's Journal* is a popular publication aimed directly at programmers, and that article brought Lua to the attention of the software industry. Among several messages that we received right after that publication was one sent in January 1997 by Bret Mogilefsky, who was the lead programmer of Grim Fandango, an adventure game then under development by LucasArts. Bret told us that he had read about Lua in *Dr. Dobb's* and that they planned to replace their home-brewed scripting language with Lua. Grim Fandango was released in October 1998 and in May 1999 Bret told us that “*a tremendous amount of the game was written in Lua*” (his emphasis) [38].⁷ Around that time, Bret attended a roundtable about game scripting at the Game Developers’ Conference (GDC, the main event for game programmers) and at the end he related his experience with the successful use of Lua in Grim Fandango. We know of several developers who first learned about Lua at that event. After that, Lua spread by word of mouth among game developers to become a definitely marketable skill in the game industry (see §8).

As a consequence of Lua’s international exposure, the number of messages sent to us asking questions about Lua increased substantially. To handle this traffic more efficiently, and also to start building a Lua community, so that other people could answer Lua questions, in February 1997 we created a mailing list for discussing Lua. Over 38,000 messages have been posted to this list since then. The use of Lua in many popular games has attracted many people to the list, which now has over 1200 subscribers. We have been fortunate that the Lua list is very friendly and at the same time very technical. The list has become the focal point of the Lua community and has been a source of motivation for improving Lua. All important events occur first in the mailing list: release announcements, feature requests, bug reports, etc.

The creation of a `comp.lang.lua` Usenet newsgroup was discussed twice in the list over all these years, in April 1998 and in July 1999. The conclusion both times was that the traffic in the list did not warrant the creation of a newsgroup. Moreover, most people preferred a mailing list. The creation of a newsgroup seems no longer relevant because there are several web interfaces for reading and searching the complete list archives.

⁶In November 1997, that article won the First Prize (technological category) in the II Compaq Award for Research and Development in Computer Science, a joint venture of Compaq Computer in Brazil, the Brazilian Ministry of Science and Technology, and the Brazilian Academy of Sciences.

⁷Grim Fandango mentioned Lua and PUC-Rio in its final credits. Several people at PUC-Rio first learned about Lua from that credit screen, and were surprised to learn that Brazilian software was part of a hit game. It has always bothered us that Lua is widely known abroad but has remained relatively unknown in Brazil until quite recently.

5.3 Lua 3

The fallback mechanism introduced in Lua 2.1 to support extensible semantics worked quite well but it was a global mechanism: there was only one hook for each event. This made it difficult to share or reuse code because modules that defined fallbacks for the same event could not co-exist easily. Following a suggestion by Stephan Herrmann in December 1996, in Lua 3.0 (July 1997) we solved the fallback clash problem by replacing fallbacks with *tag methods*: the hooks were attached to pairs (*event, tag*) instead of just to events. Tags had been introduced in Lua 2.1 as integer labels that could be attached to userdata (see §6.10); the intention was that C objects of the same type would be represented in Lua by userdata having the same tag. (However, Lua did not force any interpretation on tags.) In Lua 3.0 we extended tags to all values to support tag methods. The evolution of fallbacks is discussed in §6.8.

Lua 3.1 (July 1998) brought functional programming to Lua by introducing anonymous functions and function closures via “*upvalues*”. (Full lexical scoping had to wait until Lua 5.0; see §6.6.) The introduction of closures was mainly motivated by the existence of higher-order functions, such as `gsub`, which took functions as arguments. During the work on Lua 3.1, there were discussions in the mailing list about multithreading and cooperative multitasking, mainly motivated by the changes Bret Mogilefsky had made to Lua 2.5 and 3.1 alpha for Grim Fandango. No conclusions were reached, but the topic remained popular. Cooperative multitasking in Lua was finally provided in Lua 5.0 (April 2003); see §6.7.

The C API remained largely unchanged from Lua 1.0 to Lua 3.2; it worked over an implicit Lua state. However, newer applications, such as web services, needed multiple states. To mitigate this problem, Lua 3.1 introduced multiple independent Lua states that could be switched at run time. A fully reentrant API would have to wait until Lua 4.0. In the meantime, two unofficial versions of Lua 3.2 with explicit Lua states appeared: one written in 1998 by Roberto Ierusalimschy and Anna Hester based on Lua 3.2 alpha for CGILua [26], and one written in 1999 by Erik Hougaard based on Lua 3.2 final. Erik’s version was publicly available and was used in the Crazy Ivan robot. The version for CGILua was released only as part of the CGILua distribution; it never existed as an independent package.

Lua 3.2 (July 1999) itself was mainly a maintenance release; it brought no novelties except for a debug library that allowed tools to be written in Lua instead of C. Nevertheless, Lua was quite stable by then and Lua 3.2 had a long life. Because the next version (Lua 4.0) introduced a new, incompatible API, many users just stayed with Lua 3.2 and never migrated to Lua 4.0. For instance, Tecgraf never migrated to Lua 4.0, opting to move directly to Lua 5.0; many products at Tecgraf still use Lua 3.2.

5.4 Lua 4

Lua 4.0 was released in November 2000. As mentioned above, the main change in Lua 4.0 was a fully reentrant API, motivated by applications that needed multiple Lua states. Since making the API fully reentrant was already a major change, we took the opportunity and completely redesigned the API around a clear stack metaphor for exchanging values with C (see §6.9). This was first suggested by Reuben Thomas in July 2000.

Lua 4.0 also introduced a ‘for’ statement, then a top item in the wish-list of most Lua users and a frequent topic in the mailing list. We had not included a ‘for’ statement earlier because ‘while’ loops were more general. However, users complained that they kept forgetting to update the control variable at the end of ‘while’ loops, thus leading to infinite loops. Also, we could not agree on a good syntax. We considered the Modula ‘for’ too restrictive because it did not cover iterations over the elements of a table or over the lines of a file. A ‘for’ loop in the C tradition did not fit with the rest of Lua. With the introduction of closures and anonymous functions in Lua 3.1, we decided to use higher-order functions for implementing iterations. So, Lua 3.1 provided a higher-order function that iterated over a table by calling a user-supplied function over all pairs in the table. To print all pairs in a table *t*, one simply said ‘foreach(*t*, *print*)’.

In Lua 4.0 we finally designed a ‘for’ loop, in two variants: a numeric loop and a table-traversal loop (first suggested by Michael Spalinski in October 1997). These two variants covered most common loops; for a really generic loop, there was still the ‘while’ loop. Printing all pairs in a table *t* could then be done as follows:⁸

```
for k,v in t do
    print(k,v)
end
```

The addition of a ‘for’ statement was a simple one but it did change the look of Lua programs. In particular, Roberto had to rewrite many examples in his draft book on Lua programming. Roberto had been writing this book since 1998, but he could never finish it because Lua was a moving target. With the release of Lua 4.0, large parts of the book and almost all its code snippets had to be rewritten.

Soon after the release of Lua 4.0, we started working on Lua 4.1. Probably the main issue we faced for Lua 4.1 was whether and how to support multithreading, a big issue at that time. With the growing popularity of Java and Pthreads, many programmers began to consider support for multithreading as an essential feature in any programming language. However, for us, supporting multithreading in Lua posed serious questions. First, to implement multithreading in C requires primitives that are not part of ANSI C—

⁸With the introduction of ‘for’ iterators in Lua 5.0, this syntax was marked as obsolete and later removed in Lua 5.1.

although Pthreads was popular, there were (and still there are) many platforms without this library. Second, and more important, we did not (and still do not) believe in the standard multithreading model, which is preemptive concurrency with shared memory: we still think that no one can write correct programs in a language where ‘*a=a+1*’ is not deterministic.

For Lua 4.1, we tried to solve those difficulties in a typical Lua fashion: we implemented only a basic mechanism of multiple stacks, which we called *threads*. External libraries could use those Lua threads to implement multithreading, based on a support library such as Pthreads. The same mechanism could be used to implement coroutines, in the form of non-preemptive, collaborative multithreading. Lua 4.1 alpha was released in July 2001 with support for external multithreading and coroutines; it also introduced support for weak tables and featured a register-based virtual machine, with which we wanted to experiment.

The day after Lua 4.1 alpha was released, John D. Ramsdell started a big discussion in the mailing list about lexical scoping. After several dozen messages, it became clear that Lua needed full lexical scoping, instead of the upvalue mechanism adopted since Lua 3.1. By October 2001 we had come up with an efficient implementation of full lexical scoping, which we released as a work version in November 2001. (See §6.6 for a detailed discussion of lexical scoping.) That version also introduced a new hybrid representation for tables that let them be implemented as arrays when appropriate (see §6.2 for further details). Because that version implemented new basic algorithms, we decided to release it as a work version, even though we had already released an alpha version for Lua 4.1.

In February 2002 we released a new work version for Lua 4.1, with three relevant novelties: a generic ‘for’ loop based on iterator functions, metatables and metamethods as a replacement for tags and fallbacks⁹ (see §6.8), and coroutines (see §6.7). After that release, we realized that Lua 4.1 would bring too many major changes—perhaps ‘Lua 5.0’ would be a better name for the next version.

5.5 Lua 5

The final blow to the name ‘Lua 4.1’ came a few days later, during the Lua Library Design Workshop organized by Christian Lindig and Norman Ramsey at Harvard. One of the main conclusions of the workshop was that Lua needed some kind of module system. Although we had always considered that modules could be implemented using tables, not even the standard Lua libraries followed this path. We then decided to take that step for the next version.

⁹The use of ordinary Lua tables for implementing extensible semantics had already been suggested by Stephan Herrmann in December 1996, but we forgot all about it until it was suggested again by Edgar Toernig in October 2000, as part of a larger proposal, which he called ‘unified methods’. The term ‘metatable’ was suggested by Rici Lake in November 2001.

Packaging library functions inside tables had a big practical impact, because it affected any program that used at least one library function. For instance, the old `strfind` function was now called `string.find` (field ‘`find`’ in `string` library stored in the ‘`string`’ table); `openfile` became `io.open`; `sin` became `math.sin`; and so on. To make the transition easier, we provided a compatibility script that defined the old functions in terms of the new ones:

```
strfind = string.find
openfile = io.open
sin = math.sin
...
```

Nevertheless, packaging libraries in tables was a major change. In June 2002, when we released the next work version incorporating this change, we dropped the name ‘Lua 4.1’ and named it ‘Lua 5.0 work0’. Progress to the final version was steady from then on and Lua 5.0 was released in April 2003. This release froze Lua enough to allow Roberto to finish his book, which was published in December 2003 [27].

Soon after the release of Lua 5.0 we started working on Lua 5.1. The initial motivation was the implementation of incremental garbage collection in response to requests from game developers. Lua uses a traditional mark-and-sweep garbage collector, and, until Lua 5.0, garbage collection was performed atomically. As a consequence, some applications might experience potentially long pauses during garbage collection.¹⁰ At that time, our main concern was that adding the write barriers needed to implement an incremental garbage collector would have a negative impact on Lua performance. To compensate for that we tried to make the collector generational as well. We also wanted to keep the adaptive behavior of the old collector, which adjusted the frequency of collection cycles according to the total memory in use. Moreover, we wanted to keep the collector simple, like the rest of Lua.

We worked on the incremental generational garbage collector for over a year. But since we did not have access to applications with strong memory requirements (like games), it was difficult for us to test the collector in real scenarios. From March to December 2004 we released several work versions trying to get concrete feedback on the performance of the collector in real applications. We finally received reports of bizarre memory-allocation behavior, which we later managed to reproduce but not explain. In January 2005, Mike Pall, an active member of the Lua community, came up with memory-allocation graphs that explained the problem: in some scenarios, there were subtle interactions between the incremental behavior, the generational behavior, and the adaptive behavior, such that the collector “adapted”

¹⁰ Erik Hougaard reported that the Crazy Ivan robot would initially drive off course when Lua performed garbage collection (which could take a half second, but that was enough). To stay in course, they had to stop both motors and pause the robot during garbage collection.

for less and less frequent collections. Because it was getting too complicated and unpredictable, we gave up the generational aspect and implemented a simpler incremental collector in Lua 5.1.

During that time, programmers had been experimenting with the module system introduced in Lua 5.0. New packages started to be produced, and old packages migrated to the new system. Package writers wanted to know the best way to build modules. In July 2005, during the development of Lua 5.1, an international Lua workshop organized by Mark Hamburg was held at Adobe in San Jose. (A similar workshop organized by Wim Couwenberg and Daniel Silverstone was held in September 2006 at Océ in Venlo.) One of the presentations was about the novelties of Lua 5.1, and there were long discussions about modules and packages. As a result, we made a few small but significant changes in the module system. Despite our “mechanisms, not policy” guideline for Lua, we defined a set of policies for writing modules and loading packages, and made small changes to support these policies better. Lua 5.1 was released in February 2006. Although the original motivation for Lua 5.1 was incremental garbage collection, the improvement in the module system was probably the most visible change. On the other hand, that incremental garbage collection remained invisible shows that it succeeded in avoiding long pauses.

6. Feature evolution

In this section, we discuss in detail the evolution of some of the features of Lua.

6.1 Types

Types in Lua have been fairly stable. For a long time, Lua had only six basic types: nil, number, string, table, function, and userdata. (Actually, until Lua 3.0, C functions and Lua functions had different types internally, but that difference was transparent to callers.) The only real change happened in Lua 5.0, which introduced two new types: threads and booleans.

The type `thread` was introduced to represent coroutines. Like all other Lua values, threads are first-class values. To avoid creating new syntax, all primitive operations on threads are provided by a library.

For a long time we resisted introducing boolean values in Lua: nil was false and anything else was true. This state of affairs was simple and seemed sufficient for our purposes. However, nil was also used for absent fields in tables and for undefined variables. In some applications, it is important to allow table fields to be marked as false but still be seen as present; an explicit false value can be used for this. In Lua 5.0 we finally introduced boolean values `true` and `false`. Nil is still treated as false. In retrospect, it would probably have been better if nil raised an error in boolean expressions, as it does in other expressions. This would be more consistent with its role as proxy for undefined values. How-

ever, such a change would probably break many existing programs. LISP has similar problems, with the empty list representing both nil and false. Scheme explicitly represents false and treats the empty list as true, but some implementations of Scheme still treat the empty list as false.

6.2 Tables

Lua 1.1 had three syntactical constructs to create tables: ‘`@()`’, ‘`@[]`’, and ‘`@{}`’. The simplest form was ‘`@()`’, which created an empty table. An optional size could be given at creation time, as an efficiency hint. The form ‘`@[]`’ was used to create arrays, as in ‘`@[2,4,9,16,25]`’. In such tables, the keys were implicit natural numbers starting at 1. The form ‘`@{}`’ was used to create records, as in ‘`@{name="John", age=35}`’. Such tables were sets of key-value pairs in which the keys were explicit strings. A table created with any of those forms could be modified dynamically after creation, regardless of how it had been created. Moreover, it was possible to provide user functions when creating lists and records, as in ‘`@foo[]`’ or ‘`@foo{}`’. This syntax was inherited from SOL and was the expression of procedural data description, a major feature of Lua (see §2). The semantics was that a table was created and then the function was called with that table as its single argument. The function was allowed to check and modify the table at will, but its return values were ignored: the table was the final value of the expression.

In Lua 2.1, the syntax for table creation was unified and simplified: the leading ‘`@`’ was removed and the only constructor became ‘`{...}`’. Lua 2.1 also allowed mixed constructors, such as

```
grades{8.5, 6.0, 9.2; name="John", major="math"}
```

in which the array part was separated from the record part by a semicolon. Finally, ‘`foo{...}`’ became sugar for ‘`foo({...})`’. In other words, table constructors with functions became ordinary function calls. As a consequence, the function had to explicitly return the table (or whatever value it chose). Dropping the ‘`@`’ from constructors was a trivial change, but it actually changed the feel of the language, not merely its looks. Trivial changes that improve the feel of a language are not to be overlooked.

This simplification in the syntax and semantics of table constructors had a side-effect, however. In Lua 1.1, the equality operator was ‘`=`’. With the unification of table constructors in Lua 2.1, an expression like ‘`{a=3}`’ became ambiguous, because it could mean a table with either a pair (“`a`”, 3) or a pair (1, `b`), where `b` is the value of the equality ‘`a=3`’. To solve this ambiguity, in Lua 2.1 we changed the equality operator from ‘`=`’ to ‘`==`’. With this change, ‘`{a=3}`’ meant a table with the pair (“`a`”, 3), while ‘`{a==3}`’ meant a table with the pair (1, `b`).

These changes made Lua 2.1 incompatible with Lua 1.1 (hence the change in the major version number). Nevertheless, since at that time virtually all Lua users were from Tec-

graf, this was not a fatal move: existing programs were easily converted with the aid of ad-hoc tools that we wrote for this task.

The syntax for table constructors has since remained mostly unchanged, except for an addition introduced in Lua 3.1: keys in the record part could be given by any expression, by enclosing the expression inside brackets, as in ‘`{[10*x+f(y)]=47}`’. In particular, this allowed keys to be arbitrary strings, including reserved words and strings with spaces. Thus, ‘`{function=1}`’ is not valid (because ‘`function`’ is a reserved word), but ‘`{["function"]=1}`’ is valid. Since Lua 5.0, it is also possible to freely intermix the array part and the record part, and there is no need to use semicolons in table constructors.

While the syntax of tables has evolved, the semantics of tables in Lua has not changed at all: tables are still associative arrays and can store arbitrary pairs of values. However, frequently in practice tables are used solely as arrays (that is, with consecutive integer keys) or solely as records (that is, with string keys). Because tables are the only data-structuring mechanism in Lua, we have invested much effort in implementing them efficiently inside Lua’s core. Until Lua 4.0, tables were implemented as pure hash tables, with all pairs stored explicitly. In Lua 5.0 we introduced a hybrid representation for tables: every table contains a hash part and an array part, and both parts can be empty. Lua detects whether a table is being used as an array and automatically stores the values associated to integer indices in the array part, instead of adding them to the hash part [31]. This division occurs only at a low implementation level; access to table fields is transparent, even to the virtual machine. Tables automatically adapt their two parts according to their contents.

This hybrid scheme has two advantages. First, access to values with integer keys is faster because no hashing is needed. Second, and more important, the array part takes roughly half the memory it would take if it were stored in the hash part, because the keys are implicit in the array part but explicit in the hash part. As a consequence, if a table is being used as an array, it performs as an array, as long as its integer keys are densely distributed. Moreover, no memory or time penalty is paid for the hash part, because it does not even exist. Conversely, if the table is being used as a record and not as an array, then the array part is likely to be empty. These memory savings are important because it is common for a Lua program to create many small tables (e.g., when tables are used to represent objects). Lua tables also handle sparse arrays gracefully: the statement ‘`a={[1000000000]=1}`’ creates a table with a single entry in its hash part, not an array with one billion elements.

Another reason for investing effort into an efficient implementation of tables is that we can use tables for all kinds of tasks. For instance, in Lua 5.0 the standard library functions, which had existed since Lua 1.1 as global variables,

were moved to fields inside tables (see §5.5). More recently, Lua 5.1 brought a complete package and module system based on tables.

Tables play a prominent role in Lua’s core. On two occasions we have been able to replace special data structures inside the core with ordinary Lua tables: in Lua 4.0 for representing the global environment (which keeps all global variables) and in Lua 5.0 for implementing extensible semantics (see §6.8). Starting with Lua 4.0, global variables are stored in an ordinary Lua table, called the table of globals, a simplification suggested by John Belmonte in April 2000. In Lua 5.0 we replaced tags and tag methods (introduced in Lua 3.0) by metatables and metamethods. Metatables are ordinary Lua tables and metamethods are stored as fields in metatables. Lua 5.0 also introduced environment tables that can be attached to Lua functions; they are the tables where global names in Lua functions are resolved at run time. Lua 5.1 extended environment tables to C functions, userdata, and threads, thus replacing the notion of global environment. These changes simplified both the implementation of Lua and the API for Lua and C programmers, because globals and metamethods can be manipulated within Lua without the need for special functions.

6.3 Strings

Strings play a major role in scripting languages and so the facilities to create and manipulate strings are an important part of the usability of such languages.

The syntax for literal strings in Lua has had an interesting evolution. Since Lua 1.1, a literal string can be delimited by matching single or double quotes, and can contain C-like escape sequences. The use of both single and double quotes to delimit strings with the same semantics was a bit unusual at the time. (For instance, in the tradition of shell languages, Perl expands variables inside double-quoted strings, but not inside single-quoted strings.) While these dual quotes allow strings to contain one kind of quote without having to escape it, escape sequences are still needed for arbitrary text.

Lua 2.2 introduced *long strings*, a feature not present in classical programming languages, but present in most scripting languages.¹¹ Long strings can run for several lines and do not interpret escape sequences; they provide a convenient way to include arbitrary text as a string, without having to worry about its contents. However, it is not trivial to design a good syntax for long strings, especially because it is common to use them to include arbitrary program text (which may contain other long strings). This raises the question of how long strings end and whether they may nest. Until Lua 5.0, long strings were wrapped inside matching ‘[[…]]’ and could contain nested long strings. Unfortunately, the closing delimiter ‘]]’ could easily be part of a valid Lua program in an unbalanced way, as in ‘a[b[i]]’,

¹¹ ‘Long string’ is a Lua term. Other languages use terms such as ‘verbatim text’ or ‘heredoc’.

or in other contexts, such as ‘<[!CDATA[…]]>’ from XML. So, it was hard to reliably wrap arbitrary text as a long string.

Lua 5.1 introduced a new form for long strings: text delimited by matching ‘[===[…]==]’, where the number of ‘=’ characters is arbitrary (including zero). These new long strings do not nest: a long string ends as soon as a closing delimiter with the right number of ‘=’ is seen. Nevertheless, it is now easy to wrap arbitrary text, even text containing other long strings or unbalanced ‘]=…=]’ sequences: simply use an adequate number of ‘=’ characters.

6.4 Block comments

Comments in Lua are signaled by ‘--’ and continue to the end of the line. This is the simplest kind of comment, and is very effective. Several other languages use single-line comments, with different marks. Languages that use ‘--’ for comments include Ada and Haskell.

We never felt the need for multi-line comments, or block comments, except as a quick way to disable code. There was always the question of which syntax to use: the familiar ‘/* … */’ syntax used in C and several other languages does not mesh well with Lua’s single-line comments. There was also the question of whether block comments could nest or not, always a source of noise for users and of complexity for the lexer. Nested block comments happen when programmers want to ‘comment out’ some block of code, to disable it. Naturally, they expect that comments inside the block of code are handled correctly, which can only happen if block comments can be nested.

ANSI C supports block comments but does not allow nesting. C programmers typically disable code by using the C preprocessor idiom ‘#if 0 … #endif’. This scheme has the clear advantage that it interacts gracefully with existing comments in the disabled code. With this motivation and inspiration, we addressed the need for disabling blocks of code in Lua — not the need for block comments — by introducing conditional compilation in Lua 3.0 via pragmas inspired in the C preprocessor. Although conditional compilation could be used for block comments, we do not think that it ever was. During work on Lua 4.0, we decided that the support for conditional compilation was not worth the complexity in the lexer and in its semantics for the user, especially after not having reached any consensus about a full macro facility (see §7). So, in Lua 4.0 we removed support for conditional compilation and Lua remained without support for block comments.¹²

Block comments were finally introduced in Lua 5.0, in the form ‘--[[…]]’. Because they intentionally mimicked the syntax of long strings (see §6.3), it was easy to modify the lexer to support block comments. This similarity also helped users to grasp both concepts and their syntax. Block

¹² A further motivation was that by that time we had found a better way to generate and use debug information, and so the pragmas that controlled this were no longer needed. Removing conditional compilation allowed us to get rid of all pragmas.

comments can also be used to disable code: the idiom is to surround the code between two lines containing ‘`--[[`’ and ‘`--]]`’. The code inside those lines can be re-enabled by simply adding a single ‘`-`’ at the start of the first line: both lines then become harmless single-line comments.

Like long strings, block comments could nest, but they had the same problems as long strings. In particular, valid Lua code containing unbalanced ‘`]]`’s, such as ‘`a[b[i]]`’, could not be reliably commented out in Lua 5.0. The new scheme for long strings in Lua 5.1 also applies to block comments, in the form of matching ‘`--[==[[...]==]`’, and so provides a simple and robust solution for this problem.

6.5 Functions

Functions in Lua have always been first-class values. A function can be created at run time by compiling and executing a string containing its definition.¹³ Since the introduction of anonymous functions and upvalues in Lua 3.1, programmers are able to create functions at run time without resorting to compilation from text.

Functions in Lua, whether written in C or in Lua, have no declaration. At call time they accept a variable number of arguments: excess arguments are discarded and missing arguments are given the value `nil`. (This coincides with the semantics of multiple assignment.) C functions have always been able to handle a variable number of arguments. Lua 2.5 introduced *vararg* Lua functions, marked by a parameter list ending in ‘`...`’ (an experimental feature that became official only in Lua 3.0). When a vararg function was called, the arguments corresponding to the dots were collected into a table named ‘`arg`’. While this was simple and mostly convenient, there was no way to pass those arguments to another function, except by unpacking this table. Because programmers frequently want to just pass the arguments along to other functions, Lua 5.1 allows ‘`...`’ to be used in argument lists and on the right-hand side of assignments. This avoids the creation of the ‘`arg`’ table if it is not needed.

The unit of execution of Lua is called a *chunk*; it is simply a sequence of statements. A chunk in Lua is like the main program in other languages: it can contain both function definitions and executable code. (Actually, a function definition is executable code: an assignment.) At the same time, a chunk closely resembles an ordinary Lua function. For instance, chunks have always had exactly the same kind of bytecode as ordinary Lua functions. However, before Lua 5.0, chunks needed some internal magic to start executing. Chunks began to look like ordinary functions in Lua 2.2, when local variables outside functions were allowed as an undocumented feature (that became official only in Lua 3.1). Lua 2.5 allowed chunks to return values. In Lua 3.0 chunks became functions internally, except that they were executed

¹³ Some people maintain that the ability to evaluate code from text at run time and within the environment of the running program is what characterizes scripting languages.

right after being compiled; they did not exist as functions at the user level. This final step was taken in Lua 5.0, which broke the loading and execution of chunks into two steps, to provide host programmers better control for handling and reporting errors. As a consequence, in Lua 5.0 chunks became ordinary anonymous functions with no arguments. In Lua 5.1 chunks became anonymous vararg functions and thus can be passed values at execution time. Those values are accessed via the new ‘`...`’ mechanism.

From a different point of view, chunks are like modules in other languages: they usually provide functions and variables to the global environment. Originally, we did not intend Lua to be used for large-scale programming and so we did not feel the need to add an explicit notion of modules to Lua. Moreover, we felt that tables would be sufficient for building modules, if necessary. In Lua 5.0 we made that feeling explicit by packaging all standard libraries into tables. This encouraged other people to do the same and made it easier to share libraries. We now feel that Lua can be used for large-scale programming, especially after Lua 5.1 brought a package system and a module system, both based on tables.

6.6 Lexical scoping

From an early stage in the development of Lua we started thinking about first-class functions with full lexical scoping. This is an elegant construct that fits well within Lua’s philosophy of providing few but powerful constructs. It also makes Lua apt for functional programming. However, we could not figure out a reasonable implementation for full lexical scoping. Since the beginning Lua has used a simple array stack to keep activation records (where all local variables and temporaries live). This implementation had proved simple and efficient, and we saw no reason to change it. When we allow nested functions with full lexical scoping, a variable used by an inner function may outlive the function that created it, and so we cannot use a stack discipline for such variables.

Simple Scheme implementations allocate frames in the heap. Already in 1987, Dybvig [20] described how to use a stack to allocate frames, provided that those frames did not contain variables used by nested functions. His method requires that the compiler know beforehand whether a variable appears as a free variable in a nested function. This does not suit the Lua compiler because it generates code to manipulate variables as soon as it parses an expression; at that moment, it cannot know whether any variable is later used free in a nested function. We wanted to keep this design for implementing Lua, because of its simplicity and efficiency, and so could not use Dybvig’s method. For the same reason, we could not use advanced compiler techniques, such as data-flow analysis.

Currently there are several optimization strategies to avoid using the heap for frames (e.g., [21]), but they all need compilers with intermediate representations, which the Lua compiler does not use. McDermott’s proposal for stack frame allocation [36], which is explicitly addressed to inter-

preters, is the only one we know of that does not require intermediate representation for code generation. Like our current implementation [31], his proposal puts variables in the stack and moves them to the heap on demand, if they go out of scope while being used by a nested closure. However, his proposal assumes that environments are represented by association lists. So, after moving an environment to the heap, the interpreter has to correct only the list header, and all accesses to local variables automatically go to the heap. Lua uses real records as activation records, with local-variable access being translated to direct accesses to the stack plus an offset, and so cannot use McDermott's method.

For a long time those difficulties kept us from introducing nested first-class functions with full lexical scoping in Lua. Finally, in Lua 3.1 we settled on a compromise that we called *upvalues*. In this scheme, an inner function cannot access and modify external variables when it runs, but it can access the values those variables had when the function was created. Those values are called *upvalues*. The main advantage of upvalues is that they can be implemented with a simple scheme: all local variables live in the stack; when a function is created, it is wrapped in a closure containing copies of the values of the external variables used by the function. In other words, upvalues are the frozen values of external variables.¹⁴ To avoid misunderstandings, we created a new syntax for accessing upvalues: ‘%varname’. This syntax made it clear that the code was accessing the frozen value of that variable, not the variable itself. Upvalues proved to be very useful, despite being immutable. When necessary, we could simulate mutable external variables by using a table as the upvalue: although we could not change the table itself, we could change its fields. This feature was especially useful for anonymous functions passed to higher-order functions used for table traversal and pattern matching.

In December 2000, Roberto wrote in the first draft of his book [27] that “Lua has a form of proper lexical scoping through upvalues.” In July 2001 John D. Ramsdell argued in the mailing list that “a language is either lexically scoped or it is not; adding the adjective ‘proper’ to the phrase ‘lexical scoping’ is meaningless.” That message stirred us to search for a better solution and a way to implement full lexical scoping. By October 2001 we had an initial implementation of full lexical scoping and described it to the list. The idea was to access each upvalue through an indirection that pointed to the stack while the variable was in scope; at the end of the scope a special virtual machine instruction “closed” the upvalue, moving the variable’s value to a heap-allocated space and correcting the indirection to point there. Open closures (those with upvalues still pointing to the stack) were kept in a list to allow their correction and

the reuse of open upvalues. Reuse is essential to get the correct semantics. If two closures, sharing an external variable, have their own upvalues, then at the end of the scope each closure will have its own copy of the variable, but the correct semantics dictates that they should share the variable. To ensure reuse, the algorithm that created closures worked as follows: for each external variable used by the closure, it first searched the list of open closures. If it found an upvalue pointing to that external variable, it reused that upvalue; otherwise, it created a new upvalue.

Edgar Toering, an active member of the Lua community, misunderstood our description of lexical scoping. It turned out that the way he understood it was better than our original idea: instead of keeping a list of open closures, keep a list of open upvalues. Because the number of local variables used by closures is usually smaller than the number of closures using them (the first is statically limited by the program text), his solution was more efficient than ours. It was also easier to adapt to coroutines (which were being implemented at around the same time), because we could keep a separate list of upvalues for each stack. We added full lexical scoping to Lua 5.0 using this algorithm because it met all our requirements: it could be implemented with a one-pass compiler; it imposed no burden on functions that did not access external local variables, because they continued to manipulate all their local variables in the stack; and the cost to access an external local variable was only one extra indirection [31].

6.7 Coroutines

For a long time we searched for some kind of first-class continuations for Lua. This search was motivated by the existence of first-class continuations in Scheme (always a source of inspiration to us) and by demands from game programmers for some mechanism for “soft” multithreading (usually described as “some way to suspend a character and continue it later”).

In 2000, Maria Julia de Lima implemented full first-class continuations on top of Lua 4.0 alpha, as part of her Ph.D. work [35]. She used a simple approach because, like lexical scoping, smarter techniques to implement continuations were too complex compared to the overall simplicity of Lua. The result was satisfactory for her experiments, but too slow to be incorporated in a final product. Nevertheless, her implementation uncovered a problem peculiar to Lua. Since Lua is an extensible extension language, it is possible (and common) to call Lua from C and C from Lua. Therefore, at any given point in the execution of a Lua program, the current continuation usually has parts in Lua mixed with parts in C. Although it is possible to manipulate a Lua continuation (essentially by manipulating the Lua call stack), it is impossible to manipulate a C continuation within ANSI C. At that time, we did not understand this problem deeply enough. In particular, we could not figure out what the exact restrictions related to C calls were. Lima simply forbade any C calls in her implementation. Again, that solution was

¹⁴ A year later Java adopted a similar solution to allow inner classes. Instead of freezing the value of an external variable, Java insists that you can only access *final* variables in inner classes, and so ensures that the variable is frozen.

satisfactory for her experiments, but unacceptable for an official Lua version because the ease of mixing Lua code with C code is one of Lua’s hallmarks.

Unaware of this difficulty, in December 2001 Thatcher Ulrich announced in the mailing list:

I’ve created a patch for Lua 4.0 that makes calls from Lua to Lua non-recursive (i.e., ‘stackless’). This allows the implementation of a ‘sleep()’ call, which exits from the host program [...], and leaves the Lua state in a condition where the script can be resumed later via a call to a new API function, `lua_resume`.

In other words, he proposed an asymmetric coroutine mechanism, based on two primitives: `yield` (which he called `sleep`) and `resume`. His patch followed the high-level description given in the mailing list by Bret Mogilefsky on the changes made to Lua 2.5 and 3.1 to add cooperative multitasking in Grim Fandango. (Bret could not provide details, which were proprietary.)

Shortly after this announcement, during the Lua Library Design Workshop held at Harvard in February 2002, there was some discussion about first-class continuations in Lua. Some people claimed that, if first-class continuations were deemed too complex, we could implement one-shot continuations. Others argued that it would be better to implement symmetric coroutines. But we could not find a proper implementation of any of these mechanisms that could solve the difficulty related to C calls.

It took us some time to realize why it was hard to implement symmetric coroutines in Lua, and also to understand how Ulrich’s proposal, based on asymmetric coroutines, avoided our difficulties. Both one-shot continuations and symmetric coroutines involve the manipulation of full continuations. So, as long as these continuations include any C part, it is impossible to capture them (except by using facilities outside ANSI C). In contrast, an asymmetric coroutine mechanism based on `yield` and `resume` manipulates *partial* continuations: `yield` captures the continuation up to the corresponding `resume` [19]. With asymmetric coroutines, the current continuation can include C parts, as long as they are outside the partial continuation being captured. In other words, the only restriction is that we cannot yield across a C call.

After that realization, and based on Ulrich’s proof-of-concept implementation, we were able to implement asymmetrical coroutines in Lua 5.0. The main change was that the interpreter loop, which executes the instructions for the virtual machine, ceased to be recursive. In previous versions, when the interpreter loop executed a CALL instruction, it called itself recursively to execute the called function. Since Lua 5.0, the interpreter behaves more like a real CPU: when it executes a CALL instruction, it pushes some context information onto a call stack and proceeds to execute the called function, restoring the context when that function returns.

After that change, the implementation of coroutines became straightforward.

Unlike most implementations of asymmetrical coroutines, in Lua coroutines are what we call *stackfull* [19]. With them, we can implement symmetrical coroutines and even the `call/1cc` operator (*call with current one-shot continuation*) proposed for Scheme [11]. However, the use of C functions is severely restricted within these implementations.

We hope that the introduction of coroutines in Lua 5.0 marks a revival of coroutines as powerful control structures [18].

6.8 Extensible semantics

As mentioned in §5.2, we introduced extensible semantics in Lua 2.1 in the form of *fallbacks* as a general mechanism to allow the programmer to intervene whenever Lua did not know how to proceed. Fallbacks thus provided a restricted form of resumable exception handling. In particular, by using fallbacks, we could make a value respond to operations not originally meant for it or make a value of one type behave like a value of another type. For instance, we could make userdata and tables respond to arithmetic operations, userdata behave as tables, strings behave as functions, etc. Moreover, we could make a table respond to keys that were absent in it, which is fundamental for implementing inheritance. With fallbacks for table indexing and a little syntactic sugar for defining and calling methods, object-oriented programming with inheritance became possible in Lua.

Although objects, classes, and inheritance were not core concepts in Lua, they could be implemented directly in Lua, in many flavors, according to the needs of the application. In other words, Lua provided mechanisms, not policy — a tenet that we have tried to follow closely ever since.

The simplest kind of inheritance is inheritance by delegation, which was introduced by Self and adopted in other prototype-based languages such as NewtonScript and JavaScript. The code below shows an implementation of inheritance by delegation in Lua 2.1.

```
function Index(a,i)
    if i == "parent" then
        return nil
    end
    local p = a.parent
    if type(p) == "table" then
        return p[i]
    else
        return nil
    end
end
setfallback("index", Index)
```

When a table was accessed for an absent field (be it an attribute or a method), the index fallback was triggered. Inheritance was implemented by setting the index fallback to follow a chain of “parents” upwards, possibly triggering

the index fallback again, until a table had the required field or the chain ended.

After setting that index fallback, the code below printed ‘red’ even though ‘b’ did not have a ‘color’ field:

```
a=Window{x=100, y=200, color="red"}  
b=Window{x=300, y=400, parent=a}  
print(b.color)
```

There was nothing magical or hard-coded about delegation through a “parent” field. Programmers had complete freedom: they could use a different name for the field containing the parent, they could implement multiple inheritance by trying a list of parents, etc. Our decision not to hard-code any of those possible behaviors led to one of the main design concepts of Lua: *meta-mechanisms*. Instead of littering the language with lots of features, we provided ways for users to program the features themselves, in the way they wanted them, and *only* for those features they needed.

Fallbacks greatly increased the expressiveness of Lua. However, fallbacks were global handlers: there was only one function for each event that could occur. As a consequence, it was difficult to mix different inheritance mechanisms in the same program, because there was only one hook for implementing inheritance (the index fallback). While this might not be a problem for a program written by a single group on top of its own object system, it became a problem when one group tried to use code from other groups, because their visions of the object system might not be consistent with each other. Hooks for different mechanisms could be chained, but chaining was slow, complicated, error-prone, and not very polite. Fallback chaining did not encourage code sharing and reuse; in practice almost nobody did it. This made it very hard to use third-party libraries.

Lua 2.1 allowed userdata to be tagged. In Lua 3.0 we extended tags to all values and replaced fallbacks with *tag methods*. Tag methods were fallbacks that operated only on values with a given tag. This made it possible to implement independent notions of inheritance, for instance. No chaining was needed because tag methods for one tag did not affect tag methods for another tag.

The tag method scheme worked very well and lasted until Lua 5.0, when we replaced tags and tag methods by *metatables* and *metamethods*. Metatables are just ordinary Lua tables and so can be manipulated within Lua without the need for special functions. Like tags, metatables can be used to represent user-defined types with userdata and tables: all objects of the same “type” should share the same metatable. Unlike tags, metatables and their contents are naturally collected when no references remain to them. (In contrast, tags and their tag methods had to live until the end of the program.) The introduction of metatables also simplified the implementation: while tag methods had their own private representation inside Lua’s core, metatables use mainly the standard table machinery.

The code below shows the implementation of inheritance in Lua 5.0. The index metamethod replaces the index tag method and is represented by the ‘`__index`’ field in the metatable. The code makes ‘b’ inherit from ‘a’ by setting a metatable for ‘b’ whose ‘`__index`’ field points to ‘a’. (In general, index metamethods are functions, but we have allowed them to be tables to support simple inheritance by delegation directly.)

```
a=Window{x=100, y=200, color="red"}  
b=Window{x=300, y=400}  
setmetatable(b,{ __index = a })  
print(b.color)    --> red
```

6.9 C API

Lua is provided as a library of C functions and macros that allow the host program to communicate with Lua. This API between Lua and C is one of the main components of Lua; it is what makes Lua an embeddable language.

Like the rest of the language, the API has gone through many changes during Lua’s evolution. Unlike the rest of the language, however, the API design received little outside influence, mainly because there has been little research activity in this area.

The API has always been bi-directional because, since Lua 1.0, we have considered calling Lua from C and calling C from Lua equally important. Being able to call Lua from C is what makes Lua an *extension language*, that is, a language for extending applications through configuration, macros, and other end-user customizations. Being able to call C from Lua makes Lua an *extensible language*, because we can use C functions to extend Lua with new facilities. (That is why we say that Lua is an extensible extension language [30].) Common to both these aspects are two mismatches between C and Lua to which the API must adjust: static typing in C versus dynamic typing in Lua and manual memory management in C versus automatic garbage collection in Lua.

Currently, the C API solves both difficulties by using an abstract stack¹⁵ to exchange data between Lua and C. Every C function called by Lua gets a new stack frame that initially contains the function arguments. If the C function wants to return values to Lua, it pushes those values onto the stack just before returning.

Each stack slot can hold a Lua value of any type. For each Lua type that has a corresponding representation in C (e.g., strings and numbers), there are two API functions: an *injection* function, which pushes onto the stack a Lua value corresponding to the given C value; and a *projection* function, which returns a C value corresponding to the Lua value at a given stack position. Lua values that have no corresponding representation in C (e.g., tables and functions) can be manipulated via the API by using their stack positions.

¹⁵Throughout this section, ‘stack’ always means this abstract stack. Lua never accesses the C stack.

Practically all API functions get their operands from the stack and push their results onto the stack. Since the stack can hold values of any Lua type, these API functions operate with any Lua type, thus solving the typing mismatch. To prevent the collection of Lua values in use by C code, the values in the stack are never collected. When a C function returns, its Lua stack frame vanishes, automatically releasing all Lua values that the C function was using. These values will eventually be collected if no further references to them exist. This solves the memory management mismatch.

It took us a long time to arrive at the current API. To discuss how the API evolved, we use as illustration the C equivalent of the following Lua function:

```
function foo(t)
    return t.x
end
```

In words, this function receives a single parameter, which should be a table, and returns the value stored at the ‘x’ field in that table. Despite its simplicity, this example illustrates three important issues in the API: how to get parameters, how to index tables, and how to return results.

In Lua 1.0, we would write `foo` in C as follows:

```
void foo_1 (void) {
    lua_Object t = lua_getparam(1);
    lua_Object r = lua_getfield(t, "x");
    lua_pushobject(r);
}
```

Note that the required value is stored at the string index “x” because ‘`t.x`’ is syntactic sugar for ‘`t["x"]`’. Note also that all components of the API start with ‘`lua_`’ (or ‘`LUA_`’) to avoid name clashes with other C libraries.

To export this C function to Lua with the name ‘`foo`’ we would do

```
lua_register("foo", foo_1);
```

After that, `foo` could be called from Lua code just like any other Lua function:

```
t = {x = 200}
print(foo(t))      --> 200
```

A key component of the API was the type `lua_Object`, defined as follows:

```
typedef struct Object *lua_Object;
```

In words, `lua_Object` was an abstract type that represented Lua values in C opaquely. Arguments given to C functions were accessed by calling `lua_getparam`, which returned a `lua_Object`. In the example, we call `lua_getparam` once to get the table, which is supposed to be the first argument to `foo`. (Extra arguments are silently ignored.) Once the table is available in C (as a `lua_Object`), we get the value of its “x” field by calling `lua_getfield`. This value is also represented in C as a `lua_Object`, which is finally sent back to Lua by pushing it onto the stack with `lua_pushobject`.

The `stack` was another key component of the API. It was used to pass values from C to Lua. There was one push function for each Lua type with a direct representation in C: `lua_pushnumber` for numbers, `lua_pushstring` for strings, and `lua_pushnil`, for the special value nil. There was also `lua_pushobject`, which allowed C to pass back to Lua an arbitrary Lua value. When a C function returned, all values in the stack were returned to Lua as the results of the C function (functions in Lua can return multiple values).

Conceptually, a `lua_Object` was a union type, since it could refer to any Lua value. Several scripting languages, including Perl, Python, and Ruby, still use a union type to represent their values in C. The main drawback of this representation is that it is hard to design a garbage collector for the language. Without extra information, the garbage collector cannot know whether a value has a reference to it stored as a union in the C code. Without this knowledge, the collector may collect the value, making the union a dangling pointer. Even when this union is a local variable in a C function, this C function can call Lua again and trigger garbage collection.

Ruby solves this problem by inspecting the C stack, a task that cannot be done in a portable way. Perl and Python solve this problem by providing explicit reference-count functions for these union values. Once you increment the reference count of a value, the garbage collector will not collect that value until you decrement the count to zero. However, it is not easy for the programmer to keep these reference counts right. Not only is it easy to make a mistake, but it is difficult to find the error later (as anyone who has ever debugged memory leaks and dangling pointers can attest). Furthermore, reference counting cannot deal with cyclic data structures that become garbage.

Lua never provided such reference-count functions. Before Lua 2.1, the best you could do to ensure that an unanchored `lua_Object` was not collected was to avoid calling Lua whenever you had a reference to such a `lua_Object`. (As long as you could ensure that the value referred to by the union was also stored in a Lua variable, you were safe.) Lua 2.1 brought an important change: it kept track of all `lua_Object` values passed to C, ensuring that they were not collected while the C function was active. When the C function returned to Lua, then (and only then) all references to these `lua_Object` values were released, so that they could be collected.¹⁶

More specifically, in Lua 2.1 a `lua_Object` ceased to be a pointer to Lua’s internal data structures and became an index into an internal array that stored all values that had to be given to C:

```
typedef unsigned int lua_Object;
```

This change made the use of `lua_Object` reliable: while a value was in that array, it would not be collected by Lua.

¹⁶ A similar method is used by JNI to handle “local references”.

When the C function returned, its whole array was erased, and the values used by the function could be collected if possible. (This change also gave more freedom for implementing the garbage collector, because it could move objects if necessary; however, we did not follow this path.)

For simple uses, the Lua 2.1 behavior was very practical: it was safe and the C programmer did not have to worry about reference counts. Each `lua_Object` behaved like a local variable in C: the corresponding Lua value was guaranteed to be alive during the lifetime of the C function that produced it. For more complex uses, however, this simple scheme had two shortcomings that demanded extra mechanisms: sometimes a `lua_Object` value had to be locked for longer than the lifetime of the C function that produced it; sometimes it had to be locked for a shorter time.

The first of those shortcomings had a simple solution: Lua 2.1 introduced a system of *references*. The function `lua_lock` got a Lua value from the stack and returned a reference to it. This reference was an integer that could be used any time later to retrieve that value, using the `lua_getlocked` function. (There was also a `lua_unlock` function, which destroyed a reference.) With such references, it was easy to keep Lua values in non-local C variables.

The second shortcoming was more subtle. Objects stored in the internal array were released only when the function returned. If a function used too many values, it could overflow the array or cause an out-of-memory error. For instance, consider the following higher-order iterator function, which repeatedly calls a function and prints the result until the call returns nil:

```
void l_loop (void) {
    lua_Object f = lua_getparam(1);
    for (;;) {
        lua_Object res;
        lua_callfunction(f);
        res = lua_getresult(1);
        if (lua_isnil(res)) break;
        printf("%s\n", lua_getstring(res));
    }
}
```

The problem with this code was that the string returned by each call could not be collected until the end of the loop (that is, of the whole C function), thus opening the possibility of array overflow or memory exhaustion. This kind of error can be very difficult to track, and so the implementation of Lua 2.1 set a hard limit on the size of the internal array that kept `lua_Object` values alive. That made the error easier to track because Lua could say “too many objects in a C function” instead of a generic out-of-memory error, but it did not avoid the problem.

To address the problem, the API in Lua 2.1 offered two functions, `lua_beginblock` and `lua_endblock`, that created dynamic scopes (“blocks”) for `lua_Object` values;

all values created after a `lua_beginblock` were removed from the internal array at the corresponding `lua_endblock`. However, since a block discipline could not be forced onto C programmers, it was all too common to forget to use these blocks. Moreover, such explicit scope control was a little tricky to use. For instance, a naive attempt to correct our previous example by enclosing the `for` body within a block would fail: we had to call `lua_endblock` just before the `break`, too. This difficulty with the scope of Lua objects persisted through several versions and was solved only in Lua 4.0, when we redesigned the whole API. Nevertheless, as we said before, for typical uses the API was very easy to use, and most programmers never faced the kind of situation described here. More important, the API was safe. Erroneous use could produce well-defined errors, but not dangling references or memory leaks.

Lua 2.1 brought other changes to the API. One was the introduction of `lua_getsubscript`, which allowed the use of any value to index a table. This function had no explicit arguments: it got both the table and the key from the stack. The old `lua_getfield` was redefined as a macro, for compatibility:

```
#define lua_getfield(o,f) \
    (lua_pushobject(o), lua_pushstring(f), \
     lua_getsubscript())
```

(Backward compatibility of the C API is usually implemented using macros, whenever feasible.)

Despite all those changes, syntactically the API changed little from Lua 1 to Lua 2. For instance, our illustrative function `foo` could be written in Lua 2 exactly as we wrote it for Lua 1.0. The meaning of `lua_Object` was quite different, and `lua_getfield` was implemented on top of new primitive operations, but for the average user it was as if nothing had changed. Thereafter, the API remained fairly stable until Lua 4.0.

Lua 2.4 expanded the reference mechanism to support weak references. A common design in Lua programs is to have a Lua object (typically a table) acting as a proxy for a C object. Frequently the C object must know who its proxy is and so keeps a reference to the proxy. However, that reference prevents the collection of the proxy object, even when the object becomes inaccessible from Lua. In Lua 2.4, the program could create a *weak reference* to the proxy; that reference did not prevent the collection of the proxy object. Any attempt to retrieve a collected reference resulted in a special value `LUA_NOOBJECT`.

Lua 4.0 brought two main novelties in the C API: support for multiple Lua states and a virtual stack for exchanging values between C and Lua. Support for multiple, independent Lua states was achieved by eliminating all global state. Until Lua 3.0, only one Lua state existed and it was implemented using many static variables scattered throughout the code. Lua 3.1 introduced multiple independent Lua states; all static variables were collected into a single C struct. An

API function was added to allow switching states, but only one Lua state could be active at any moment. All other API functions operated over the active Lua state, which remained implicit and did not appear in the calls. Lua 4.0 introduced explicit Lua states in the API. This created a big incompatibility with previous versions.¹⁷ All C code that communicated with Lua (in particular, all C functions registered to Lua) had to be changed to include an explicit state argument in calls to the C API. Since all C functions had to be rewritten anyway, we took this opportunity and made another major change in the C–Lua communication in Lua 4.0: we replaced the concept of `lua_Object` by an explicit virtual stack used for all communication between Lua and C in both directions. The stack could also be used to store temporary values.

In Lua 4.0, our `foo` example could be written as follows:

```
int foo_1 (lua_State *L) {
    lua_pushstring(L, "x");
    lua_gettable(L, 1);
    return 1;
}
```

The first difference is the function signature: `foo_1` now receives a Lua state on which to operate and returns the number of values returned by the function in the stack. In previous versions, all values left in the stack when the function ended were returned to Lua. Now, because the stack is used for all operations, it can contain intermediate values that are not to be returned, and so the function needs to tell Lua how many values in the stack to consider as return values. Another difference is that `lua_getparam` is no longer needed, because function arguments come in the stack when the function starts and can be directly accessed by their index, like any other stack value.

The last difference is the use of `lua_gettable`, which replaced `lua_getsubscript` as the means to access table fields. `lua_gettable` receives the table to be indexed as a stack position (instead of as a Lua object), pops the key from the top of the stack, and pushes the result. Moreover, it leaves the table in the same stack position, because tables are frequently indexed repeatedly. In `foo_1`, the table used by `lua_gettable` is at stack position 1, because it is the first argument to that function, and the key is the string "`x`", which needs to be pushed onto the stack before calling `lua_gettable`. That call replaces the key in the stack with the corresponding table value. So, after `lua_gettable`, there are two values in the stack: the table at position 1 and the result of the indexing at position 2, which is the top of the stack. The C function returns 1 to tell Lua to use that top value as the single result returned by the function.

To further illustrate the new API, here is an implementation of our loop example in Lua 4.0:

```
int l_loop (lua_State *L) {
    for (;;) {
        lua_pushvalue(L, 1);
        lua_call(L, 0, 1);
        if (lua_isnil(L, -1)) break;
        printf("%s\n", lua_tostring(L, -1));
        lua_pop(L, 1);
    }
    return 0;
}
```

To call a Lua function, we push it onto the stack and then push its arguments, if any (none in the example). Then we call `lua_call`, telling how many arguments to get from the stack (and therefore implicitly also telling where the function is in the stack) and how many results we want from the call. In the example, we have no arguments and expect one result. The `lua_call` function removes the function and its arguments from the stack and pushes back exactly the requested number of results. The call to `lua_pop` removes the single result from the stack, leaving the stack at the same level as at the beginning of the loop. For convenience, we can index the stack from the bottom, with positive indices, or from the top, with negative indices. In the example, we use index `-1` in `lua_isnil` and `lua_tostring` to refer to the top of the stack, which contains the function result.

With hindsight, the use of a single stack in the API seems an obvious simplification, but when Lua 4.0 was released many users complained about the complexity of the new API. Although Lua 4.0 had a much cleaner conceptual model for its API, the direct manipulation of the stack requires some thought to get right. Many users were content to use the previous API without any clear conceptual model of what was going on behind the scenes. Simple tasks did not require a conceptual model at all and the previous API worked quite well for them. More complex tasks often broke whatever private models users had, but most users never programmed complex tasks in C. So, the new API was seen as too complex at first. However, such skepticism gradually vanished, as users came to understand and value the new model, which proved to be simpler and much less error-prone.

The possibility of multiple states in Lua 4.0 created an unexpected problem for the reference mechanism. Previously, a C library that needed to keep some object fixed could create a reference to the object and store that reference in a global C variable. In Lua 4.0, if a C library was to work with several states, it had to keep an individual reference for each state and so could not keep the reference in a global C variable. To solve this difficulty, Lua 4.0 introduced the *registry*, which is simply a regular Lua table available to C only. With the registry, a C library that wants to keep a Lua object can choose a unique key and associate the object with this key in the registry. Because each independent Lua state has its own registry, the C library can use the same key in each state to manipulate the corresponding object.

¹⁷ We provided a module that emulated the 3.2 API on top of the 4.0 API, but we do not think it was used much.

We could quite easily implement the original reference mechanism on top of the registry by using integer keys to represent references. To create a new reference, we just find an unused integer key and store the value at that key. Retrieving a reference becomes a simple table access. However, we could not implement weak references using the registry. So, Lua 4.0 kept the previous reference mechanism. In Lua 5.0, with the introduction of weak tables in the language, we were finally able to eliminate the reference mechanism from the core and move it to a library.

The C API has slowly evolved toward completeness. Since Lua 4.0, all standard library functions can be written using only the C API. Until then, Lua had a number of built-in functions (from 7 in Lua 1.1 to 35 in Lua 3.2), most of which could have been written using the C API but were not because of a perceived need for speed. A few built-in functions could not have been written using the C API because the C API was not complete. For instance, until Lua 3.2 it was not possible to iterate over the contents of a table using the C API, although it was possible to do it in Lua using the built-in function `next`. The C API is not yet complete and not everything that can be done in Lua can be done in C; for instance, the C API lacks functions for performing arithmetic operations on Lua values. We plan to address this issue in the next version.

6.10 Userdata

Since its first version, an important feature of Lua has been its ability to manipulate C data, which is provided by a special Lua data type called *userdata*. This ability is an essential component in the extensibility of Lua.

For Lua programs, the userdata type has undergone no changes at all throughout Lua's evolution: although userdata are first-class values, userdata is an opaque type and its only valid operation in Lua is equality test. Any other operation over userdata (creation, inspection, modification) must be provided by C functions.

For C functions, the userdata type has undergone several changes in Lua's evolution. In Lua 1.0, a userdata value was a simple `void*` pointer. The main drawback of this simplicity was that a C library had no way to check whether a userdata was valid. Although Lua code cannot create userdata values, it can pass userdata created by one library to another library that expects pointers to a different structure. Because C functions had no mechanisms to check this mismatch, the result of this pointer mismatch was usually fatal to the application. We have always considered it unacceptable for a Lua program to be able to crash the host application. Lua should be a safe language.

To overcome the pointer mismatch problem, Lua 2.1 introduced the concept of *tags* (which would become the seed for *tag methods* in Lua 3.0). A tag was simply an arbitrary integer value associated with a userdata. A userdata's tag could only be set once, when the userdata was created. Provided that each C library used its own exclusive tag, C code could

easily ensure that a userdata had the expected type by checking its tag. (The problem of how a library writer chose a tag that did not clash with tags from other libraries remained open. It was only solved in Lua 3.0, which provided tag management via `lua_newtag`.)

A bigger problem with Lua 2.1 was the management of C resources. More often than not, a userdata pointed to a dynamically allocated structure in C, which had to be freed when its corresponding userdata was collected in Lua. However, userdata were values, not objects. As such, they were not collected (in the same way that numbers are not collected). To overcome this restriction, a typical design was to use a table as a proxy for the C structure in Lua, storing the actual userdata in a predefined field of the proxy table. When the table was collected, its finalizer would free the corresponding C structure.

This simple solution created a subtle problem. Because the userdata was stored in a regular field of the proxy table, a malicious user could tamper with it from within Lua. Specifically, a user could make a copy of the userdata and use the copy after the table was collected. By that time, the corresponding C structure had been destroyed, making the userdata a dangling pointer, with disastrous results. To improve the control of the life cycle of userdata, Lua 3.0 changed userdata from values to objects, subject to garbage collection. Users could use the userdata finalizer (the garbage-collection tag method) to free the corresponding C structure. The correctness of Lua's garbage collector ensured that a userdata could not be used after being collected.

However, userdata as objects created an identity problem. Given a userdata, it is trivial to get its corresponding pointer, but frequently we need to do the reverse: given a C pointer, we need to get its corresponding userdata.¹⁸ In Lua 2, two userdata with the same pointer and the same tag would be equal; equality was based on their values. So, given the pointer and the tag, we had the userdata. In Lua 3, with userdata being objects, equality was based on identity: two userdata were equal only when they were the *same* userdata (that is, the same object). Each userdata created was different from all others. Therefore, a pointer and a tag would not be enough to get the corresponding userdata.

To solve this difficulty, and also to reduce incompatibilities with Lua 2, Lua 3 adopted the following semantics for the operation of pushing a userdata onto the stack: if Lua already had a userdata with the given pointer and tag, then that userdata was pushed on the stack; otherwise, a new userdata was created and pushed on the stack. So, it was easy for C code to translate a C pointer to its corresponding userdata in Lua. (Actually, the C code could be the same as it was in Lua 2.)

¹⁸ A typical scenario for this need is the handling of callbacks in a GUI toolkit. The C callback associated with a widget gets only a pointer to the widget, but to pass this callback to Lua we need the userdata that represents that widget in Lua.

However, Lua 3 behavior had a major drawback: it combined into a single primitive (`lua_pushuserdata`) two basic operations: userdata searching and userdata creation. For instance, it was impossible to check whether a given C pointer had a corresponding userdata without creating that userdata. Also, it was impossible to create a new userdata regardless of its C pointer. If Lua already had a userdata with that value, no new userdata would be created.

Lua 4 mitigated that drawback by introducing a new function, `lua_newuserdata`. Unlike `lua_pushuserdata`, this function always created a new userdata. Moreover, what was more important at that time, those userdata were able to store arbitrary C data, instead of pointers only. The user would tell `lua_newuserdata` the amount memory to be allocated and `lua_newuserdata` returned a pointer to the allocated area. By having Lua allocate memory for the user, several common tasks related to userdata were simplified. For instance, C code did not need to handle memory-allocation errors, because they were handled by Lua. More important, C code did not need to handle memory deallocation: memory used by such userdata was released by Lua automatically, when the userdata was collected.

However, Lua 4 still did not offer a nice solution to the search problem (i.e., finding a userdata given its C pointer). So, it kept the `lua_pushuserdata` operation with its old behavior, resulting in a hybrid system. It was only in Lua 5 that we removed `lua_pushuserdata` and dissociated userdata creation and searching. Actually, Lua 5 removed the searching facility altogether. Lua 5 also introduced *light userdata*, which store plain C pointer values, exactly like regular userdata in Lua 1. A program can use a weak table to associate C pointers (represented as light userdata) to its corresponding “heavy” userdata in Lua.

As is usual in the evolution of Lua, userdata in Lua 5 is more flexible than it was in Lua 4; it is also simpler to explain and simpler to implement. For simple uses, which only require storing a C structure, userdata in Lua 5 is trivial to use. For more complex needs, such as those that require mapping a C pointer back to a Lua userdata, Lua 5 offers the mechanisms (light userdata and weak tables) for users to implement strategies suited to their applications.

6.11 Reflectivity

Since its very first version Lua has supported some reflective facilities. A major reason for this support was the proposed use of Lua as a configuration language to replace SOL. As described in §4, our idea was that the programmer could use the language itself to write type-checking routines, if needed.

For instance, if a user wrote something like

```
T = @track{ y=9, x=10, id="1992-34" }
```

we wanted to be able to check that the track did have a `y` field and that this field was a number. We also wanted to be able to check that the track did not have extraneous fields

(possibly to catch typing mistakes). For these two tasks, we needed access to the type of a Lua value and a mechanism to traverse a table and visit all its pairs.

Lua 1.0 provided the needed functionality with only two functions, which still exist: `type` and `next`. The `type` function returns a string describing the type of any given value (“`number`”, “`nil`”, “`table`”, etc.). The `next` function receives a table and a key and returns a “`next`” key in the table (in an arbitrary order). The call `next(t, nil)` returns a “`first`” key. With `next` we can traverse a table and process all its pairs. For instance, the following code prints all pairs in a table `t`:¹⁹

```
k = next(t, nil)
while k do
    print(k, t[k])
    k = next(t, k)
end
```

Both these functions have a simple implementation: `type` checks the internal tag of the given value and returns the corresponding string; `next` finds the given key in the table and then goes to the next key, following the internal table representation.

In languages like Java and Smalltalk, reflection must reify concepts like classes, methods, and instance variables. Moreover, that reification demands new concepts like metaclasses (the class of a reified class). Lua needs nothing like that. In Lua, most facilities provided by the Java reflective package come for free: classes and modules are tables, methods are functions. So, Lua does not need any special mechanism to reify them; they are plain program values. Similarly, Lua does not need special mechanisms to build method calls at run time (because functions are first-class values and Lua’s parameter-passing mechanism naturally supports calling a function with a variable number of arguments), and it does not need special mechanisms to access a global variable or an instance variable given its name (because they are regular table fields).²⁰

7. Retrospect

In this section we give a brief critique of Lua’s evolutionary process, discussing what has worked well, what we regret, and what we do not really regret but could have done differently.

One thing that has worked really well was the early decision (made in Lua 1.0) to have tables as the sole data-structuring mechanism in Lua. Tables have proved to be powerful and efficient. The central role of tables in the language and in its implementation is one of the main character-

¹⁹ Although this code still works, the current idiom is ‘`for k,v in pairs(t) do print(k,v) end`’.

²⁰ Before Lua 4.0, global variables were stored in a special data structure inside the core, and we provided a `nextvar` function to traverse it. Since Lua 4.0, global variables are stored in a regular Lua table and `nextvar` is no longer needed.

istics of Lua. We have resisted user pressure to include other data structures, mainly “real” arrays and tuples, first by being stubborn, but also by providing tables with an efficient implementation and a flexible design. For instance, we can represent a set in Lua by storing its elements as indices of a table. This is possible only because Lua tables accept any value as index.

Another thing that has worked well was our insistence on portability, which was initially motivated by the diverse platforms of Tecgraf’s clients. This allowed Lua to be compiled for platforms we had never dreamed of supporting. In particular, Lua’s portability is one of the reasons that Lua has been widely adopted for developing games. Restricted environments, such as game consoles, tend not to support the complete semantics of the full standard C library. By gradually reducing the dependency of Lua’s core on the standard C library, we are moving towards a Lua core that requires only a free-standing ANSI C implementation. This move aims mainly at embedding flexibility, but it also increases portability. For instance, since Lua 3.1 it is easy to change a few macros in the code to make Lua use an application-specific memory allocator, instead of relying on `malloc` and friends. Starting with Lua 5.1, the memory allocator can be provided dynamically when creating a Lua state.

With hindsight, we consider that being raised by a small committee has been very positive for the evolution of Lua. Languages designed by large committees tend to be too complicated and never quite fulfill the expectations of their sponsors. Most successful languages are raised rather than designed. They follow a slow bottom-up process, starting as a small language with modest goals. The language evolves as a consequence of actual feedback from real users, from which design flaws surface and new features that are actually useful are identified. This describes the evolution of Lua quite well. We listen to users and their suggestions, but we include a new feature in Lua only when all three of us agree; otherwise, it is left for the future. It is much easier to add features later than to remove them. This development process has been essential to keep the language simple, and simplicity is our most important asset. Most other qualities of Lua—speed, small size, and portability—derive from its simplicity.

Since its first version Lua has had real users, that is, users others than ourselves, who care not about the language itself but only about how to use it productively. Users have always given important contributions to the language, through suggestions, complaints, use reports, and questions. Again, our small committee plays an important role in managing this feedback: its structure gives us enough inertia to listen closely to users without having to follow all their suggestions.

Lua is best described as a closed-development, open-source project. This means that, even though the source code is freely available for scrutiny and adaption, Lua is

not developed in a collaborative way. We do accept user suggestions, but never their code verbatim. We always try to do our own implementation.

Another unusual aspect of Lua’s evolution has been our handling of incompatible changes. For a long time we considered simplicity and elegance more important than compatibility with previous versions. Whenever an old feature was superseded by a new one, we simply removed the old feature. Frequently (but not always), we provided some sort of compatibility aid, such as a compatibility library, a conversion script, or (more recently) compile-time options to preserve the old feature. In any case, the user had to take some measures when moving to a new version.

Some upgrades were a little traumatic. For instance, Tecgraf, Lua’s birthplace, never upgraded from Lua 3.2 to Lua 4.0 because of the big changes in the API. Currently, a few Tecgraf programs have been updated to Lua 5.0, and new programs are written in this version, too. But Tecgraf still has a large body of code in Lua 3.2. The small size and simplicity of Lua alleviates this problem: it is easy for a project to keep to an old version of Lua, because the project group can do its own maintenance of the code, when necessary.

We do not really regret this evolution style. Gradually, however, we have become more conservative. Not only is our user and code base much larger than it once was, but also we feel that Lua as a language is much more mature.

We should have introduced booleans from the start, but we wanted to start with the simplest possible language. Not introducing booleans from the start had a few unfortunate side-effects. One is that we now have two false values: `nil` and `false`. Another is that a common protocol used by Lua functions to signal errors to their callers is to return `nil` followed by an error message. It would have been better if `false` had been used instead of `nil` in that case, with `nil` being reserved for its primary role of signaling the absence of any useful value.

Automatic coercion of strings to numbers in arithmetic operations, which we took from Awk, could have been omitted. (Coercion of numbers to strings in string operations is convenient and less troublesome.)

Despite our “mechanisms, not policy” rule—which we have found valuable in guiding the evolution of Lua—we should have provided a precise set of policies for modules and packages earlier. The lack of a common policy for building modules and installing packages prevents different groups from sharing code and discourages the development of a community code base. Lua 5.1 provides a set of policies for modules and packages that we hope will remedy this situation.

As mentioned in §6.4, Lua 3.0 introduced support for conditional compilation, mainly motivated to provide a means to disable code. We received many requests for enhancing conditional compilation in Lua, even by people who did not

use it! By far the most popular request was for a full macro processor like the C preprocessor. Providing such a macro processor in Lua would be consistent with our general philosophy of providing extensible mechanisms. However, we would like it to be programmable in Lua, not in some other specialized language. We did not want to add a macro facility directly into the lexer, to avoid bloating it and slowing compilation. Moreover, at that time the Lua parser was not fully reentrant, and so there was no way to call Lua from within the lexer. (This restriction was removed in Lua 5.1.) So endless discussions ensued in the mailing list and within the Lua team. But no consensus was ever reached and no solution emerged. We still have not completely dismissed the idea of providing Lua with a macro system: it would give Lua extensible syntax to go with extensible semantics.

8. Conclusion

Lua has been used successfully in many large companies, such as Adobe, Bombardier, Disney, Electronic Arts, Intel, LucasArts, Microsoft, Nasa, Olivetti, and Philips. Many of these companies have shipped Lua embedded into commercial products, often exposing Lua scripts to end users.

Lua has been especially successful in games. It was said recently that “Lua is rapidly becoming the de facto standard for game scripting” [37]. Two informal polls [5, 6] conducted by gamedev.net (an important site for game programmers) in September 2003 and in June 2006 showed Lua as the most popular scripting language for game development. Roundtables dedicated to Lua in game development were held at GDC in 2004 and 2006. Many famous games use Lua: Baldur’s Gate, Escape from Monkey Island, FarCry, Grim Fandango, Homeworld 2, Illarion, Impossible Creatures, Psychonauts, The Sims, World of Warcraft. There are two books on game development with Lua [42, 25], and several other books on game development devote chapters to Lua [23, 44, 41, 24].

The wide adoption of Lua in games came as a surprise to us. We did not have game development as a target for Lua. (Tecgraf is mostly concerned with scientific software.) With hindsight, however, that success is understandable because all the features that make Lua special are important in game development:

Portability: Many games run on non-conventional platforms, such as game consoles, that need special development tools. An ANSI C compiler is all that is needed to build Lua.

Ease of embedding: Games are demanding applications. They need both performance, for its graphics and simulations, and flexibility, for the creative staff. Not by chance, many games are coded in (at least) two languages, one for scripting and the other for coding the engine. Within that framework, the ease of integrating Lua with another

language (mainly C++, in the case of games) is a big advantage.

Simplicity: Most game designers, scripters and level writers are not professional programmers. For them, a language with simple syntax and simple semantics is particularly important.

Efficiency and small size: Games are demanding applications; the time allotted to running scripts is usually quite small. Lua is one of the fastest scripting languages [1]. Game consoles are restricted environments. The script interpreter should be parsimonious with resources. The Lua core takes about 100K.

Control over code: Unlike most other software enterprises, game production involves little evolution. In many cases, once a game has been released, there are no updates or new versions, only new games. So, it is easier to risk using a new scripting language in a game. Whether the scripting language will evolve or how it will evolve is not a crucial point for game developers. All they need is the version they used in the game. Since they have complete access to the source code of Lua, they can simply keep the same Lua version forever, if they so choose.

Liberal license: Most commercial games are not open source. Some game companies even refuse to use any kind of open-source code. The competition is hard, and game companies tend to be secretive about their technologies. For them, a liberal license like the Lua license is quite convenient.

Coroutines: It is easier to script games if the scripting language supports multitasking because a character or activity can be suspended and resumed later. Lua supports cooperative multitasking in the form of coroutines [14].

Procedural data files: Lua’s original design goal of providing powerful data-description facilities allows games to use Lua for data files, replacing special-format textual data files with many benefits, especially homogeneity and expressiveness.

Acknowledgments

Lua would never be what it is without the help of many people. Everyone at Tecgraf has contributed in different forms — using the language, discussing it, disseminating it outside Tecgraf. Special thanks go to Marcelo Gattass, head of Tecgraf, who always encouraged us and gave us complete freedom over the language and its implementation. Lua is no longer a Tecgraf product but it is still developed inside PUC-Rio, in the LabLua laboratory created in May 2004.

Without users Lua would be just yet another language, destined to oblivion. Users and their uses are the ultimate test for a language. Special thanks go to the members of our mailing list, for their suggestions, complaints, and patience. The mailing list is relatively small, but it is very friendly and

contains some very strong technical people who are not part of the Lua team but who generously share their expertise with the whole community.

We thank Norman Ramsey for suggesting that we write a paper on Lua for HOPL III and for making the initial contacts with the conference chairs. We thank Julia Lawall for thoroughly reading several drafts of this paper and for carefully handling this paper on behalf of the HOPL III committee. We thank Norman Ramsey, Julia Lawall, Brent Hailpern, Barbara Ryder, and the anonymous referees for their detailed comments and helpful suggestions.

We also thank André Carregal, Anna Hester, Bret Mogilefsky, Bret Victor, Daniel Collins, David Burgess, Diego Nehab, Eric Raible, Erik Hougaard, Gavin Wraith, John Belmonte, Mark Hamburg, Peter Sommerfeld, Reuben Thomas, Stephan Herrmann, Steve Dekorte, Taj Khattra, and Thatcher Ulrich for complementing our recollection of the historical facts and for suggesting several improvements to the text. Katrina Avery did a fine copy-editing job.

Finally, we thank PUC-Rio, IMPA, and CNPq for their continued support of our work on Lua, and FINEP and Microsoft Research for supporting several projects related to Lua.

References

- [1] The computer language shootout benchmarks. <http://shootout.alioth.debian.org/>.
- [2] Lua projects. <http://www.lua.org/uses.html>.
- [3] The MIT license. <http://www.opensource.org/licenses/mit-license.html>.
- [4] Timeline of programming languages. http://en.wikipedia.org/wiki/Timeline_of_programming_languages.
- [5] Which language do you use for scripting in your game engine? <http://www.gamedev.net/gdpolls/viewpoll.asp?ID=163>, Sept. 2003.
- [6] Which is your favorite embeddable scripting language? <http://www.gamedev.net/gdpolls/viewpoll.asp?ID=788>, June 2006.
- [7] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2000.
- [8] G. Bell, R. Carey, and C. Marrin. The Virtual Reality Modeling Language Specification—Version 2.0. <http://www.vrml.org/VRML2.0/FINAL/>, Aug. 1996. (ISO/IEC CD 14772).
- [9] J. Bentley. Programming pearls: associative arrays. *Communications of the ACM*, 28(6):570–576, 1985.
- [10] J. Bentley. Programming pearls: little languages. *Communications of the ACM*, 29(8):711–721, 1986.
- [11] C. Bruggeman, O. Waddell, and R. K. Dybvig. Representing control in the presence of one-shot continuations. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 99–107, 1996.
- [12] W. Celes, L. H. de Figueiredo, and M. Gattass. EDG: uma ferramenta para criação de interfaces gráficas interativas. In *Proceedings of SIBGRAPI '95 (Brazilian Symposium on Computer Graphics and Image Processing)*, pages 241–248, 1995.
- [13] B. Davis, A. Beatty, K. Casey, D. Gregg, and J. Waldron. The case for virtual register machines. In *Proceedings of the 2003 Workshop on Interpreters, Virtual Machines and Emulators*, pages 41–49. ACM Press, 2003.
- [14] L. H. de Figueiredo, W. Celes, and R. Ierusalimschy. Programming advanced control mechanisms with Lua coroutines. In *Game Programming Gems 6*, pages 357–369. Charles River Media, 2006.
- [15] L. H. de Figueiredo, R. Ierusalimschy, and W. Celes. The design and implementation of a language for extending applications. In *Proceedings of XXI SEMISH (Brazilian Seminar on Software and Hardware)*, pages 273–284, 1994.
- [16] L. H. de Figueiredo, R. Ierusalimschy, and W. Celes. Lua: an extensible embedded language. *Dr. Dobb's Journal*, 21(12):26–33, Dec. 1996.
- [17] L. H. de Figueiredo, C. S. Souza, M. Gattass, and L. C. G. Coelho. Geração de interfaces para captura de dados sobre desenhos. In *Proceedings of SIBGRAPI '92 (Brazilian Symposium on Computer Graphics and Image Processing)*, pages 169–175, 1992.
- [18] A. de Moura, N. Rodriguez, and R. Ierusalimschy. Coroutines in Lua. *Journal of Universal Computer Science*, 10(7):910–925, 2004.
- [19] A. L. de Moura and R. Ierusalimschy. Revisiting coroutines. MCC 15/04, PUC-Rio, 2004.
- [20] R. K. Dybvig. *Three Implementation Models for Scheme*. PhD thesis, Department of Computer Science, University of North Carolina at Chapel Hill, 1987. Technical Report #87-011.
- [21] M. Feeley and G. Lapalme. Closure generation based on viewing LAMBDA as EPSILON plus COMPILE. *Journal of Computer Languages*, 17(4):251–267, 1992.
- [22] T. G. Gorham and R. Ierusalimschy. Um sistema de depuração reflexivo para uma linguagem de extensão. In *Anais do I Simpósio Brasileiro de Linguagens de Programação*, pages 103–114, 1996.
- [23] T. Gutschmidt. *Game Programming with Python, Lua, and Ruby*. Premier Press, 2003.
- [24] M. Harmon. Building Lua into games. In *Game Programming Gems 5*, pages 115–128. Charles River Media, 2005.
- [25] J. Heiss. *Lua Scripting für Spieleprogrammierer. Hit the Ground with Lua*. Stefan Zerbst, Dec. 2005.
- [26] A. Hester, R. Borges, and R. Ierusalimschy. Building flexible and extensible web applications with Lua. *Journal of Universal Computer Science*, 4(9):748–762, 1998.
- [27] R. Ierusalimschy. *Programming in Lua*. Lua.org, 2003.
- [28] R. Ierusalimschy. *Programming in Lua*. Lua.org, 2nd edition, 2006.

- [29] R. Ierusalimschy, W. Celes, L. H. de Figueiredo, and R. de Souza. Lua: uma linguagem para customização de aplicações. In *VII Simpósio Brasileiro de Engenharia de Software — Caderno de Ferramentas*, page 55, 1993.
- [30] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes. Lua: an extensible extension language. *Software: Practice & Experience*, 26(6):635–652, 1996.
- [31] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes. The implementation of Lua 5.0. *Journal of Universal Computer Science*, 11(7):1159–1176, 2005.
- [32] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes. *Lua 5.1 Reference Manual*. Lua.org, 2006.
- [33] K. Jung and A. Brown. *Beginning Lua Programming*. Wrox, 2007.
- [34] L. Lamport. *TeX: A Document Preparation System*. Addison-Wesley, 1986.
- [35] M. J. Lima and R. Ierusalimschy. Continuações em Lua. In *VI Simpósio Brasileiro de Linguagens de Programação*, pages 218–232, June 2002.
- [36] D. McDermott. An efficient environment allocation scheme in an interpreter for a lexically-scoped LISP. In *ACM conference on LISP and functional programming*, pages 154–162, 1980.
- [37] I. Millington. *Artificial Intelligence for Games*. Morgan Kaufmann, 2006.
- [38] B. Mogilefsky. Lua in Grim Fandango. <http://www.grimfandango.net/?page=articles&pagenumber=2>, May 1999.
- [39] Open Software Foundation. *OSF/Motif Programmer's Guide*. Prentice-Hall, Inc., 1991.
- [40] J. Ousterhout. Tcl: an embeddable command language. In *Proc. of the Winter 1990 USENIX Technical Conference*. USENIX Association, 1990.
- [41] D. Sanchez-Crespo. *Core Techniques and Algorithms in Game Programming*. New Riders Games, 2003.
- [42] P. Schuytema and M. Manyen. *Game Development with Lua*. Delmar Thomson Learning, 2005.
- [43] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: an annotated bibliography. *SIGPLAN Notices*, 35(6):26–36, 2000.
- [44] A. Varanese. *Game Scripting Mastery*. Premier Press, 2002.

Modula-2 and Oberon

Niklaus Wirth

ETH Zurich

wirth@inf.ethz.ch

Abstract

This is an account of the development of the languages Modula-2 and Oberon. Together with their ancestors ALGOL 60 and Pascal they form a family called Algol-like languages. Pascal (1970) reflected the ideas of structured programming, Modula-2 (1979) added those of modular system design, and Oberon (1988) catered to the object-oriented style. Thus they mirror the essential programming paradigms of the past decades. Here the major language properties are outlined, followed by an account of the respective implementation efforts. The conditions and the environments in which the languages were created are elucidated. We point out that simplicity of design was the most essential guiding principle. Clarity of concepts, economy of features, efficiency and reliability of implementations were its consequences.

Categories and Subject Descriptors D.3.3 [Programming Languages]: Language Constructs and Features – abstract data types, *classes and objects, modules*.

General Terms Design, Reliability, Languages.

1. Background

In the middle of the 1970s, the computing scene evolved around large computers. Programmers predominantly used time-shared "main frames" remotely via low-bandwidth (1200 b/s) lines and simple ("dumb") terminals displaying 24 lines of up to 80 characters. Accordingly, interactivity was severely limited, and program development and testing was a time-consuming process. Yet, the power of computers – albeit tiny in comparison with modern devices – had grown considerably over the previous decade. Therefore the complexity of tasks, and thus that of programs, had grown likewise. The notion of parallel processes had become a concern and made programming even more difficult. The limit of our intellectual capability seemed to be reached, and a noteworthy conference in 1968 gave birth to the term software crisis [1, p.120].

Small wonder, then, that hopes rested on the advent of better tools. They were seen in new programming languages, symbolic debuggers, and team management. Dijkstra put the emphasis on better education. Already in the mid-1960s he had outlined his discipline of *structured programming* [3], and the language Pascal followed his ideas and represented an incarnation of a *structured language* [2]. But the dominating languages were FORTRAN in

Permission to make digital/hard copy of part of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date of appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Permission may be requested from the Publications Dept., ACM, Inc., 2 Penn Plaza, New York, NY 11201-0701, USA, fax:+1(212) 869-0481, permissions@acm.org

©2007 ACM 978-1-59593-766-7/2007/06-ART3 \$5.00

DOI 10.1145/1238844.1238847

<http://doi.acm.org/10.1145/1238844.1238847>

scientific circles and COBOL in business data processing. IBM's PL/I was slowly gaining acceptance. It tried to unite the disparate worlds of scientific and business applications. Some further, "esoteric" languages were popular in academia, for example *Lisp* and its extensions, which dominated the AI culture with its list-processing facilities.

However, none of the available languages was truly suitable for handling the ever-growing complexity of computing tasks. FORTRAN and COBOL lacked a pervasive concept of data types like that of Pascal; and Pascal lacked a facility for piecewise compilation, and thus for program libraries. PL/I offered everything to a certain degree. Therefore it was bulky and hard to master. The fact remained that none of the available languages was truly satisfactory.

I was fortunate to be able to spend a sabbatical year at the new Xerox Research Laboratory in Palo Alto during this time. There, on the personal workstation *Alto*, I encountered the language *Mesa*, which appeared to be the appropriate language for programming large systems. It stemmed from Pascal [2, 38], and hence had adopted a strictly static typing scheme. But one was also allowed to develop parts of a system – called modules – independently, and to bind them through the linking loader into a consistent whole. This alone was nothing new. It was new, however, for strongly typed languages, guaranteeing type consistency between the linked modules. Therefore, compilation of modules was not called independent, but *separate compilation*. We will return to this topic later.

As Pascal had been Mesa's ancestor, Mesa served as Modula-2's guideline. Mesa had not only adopted Pascal's style and concepts, but also most of its facilities, to which were added many more, as they all seemed either necessary or convenient for system programming. The result was a large language, difficult to fully master, and more so to implement. Making a virtue out of necessity, I simplified Mesa, retaining what seemed essential and preserving the appearance of Pascal. The guiding idea was to construct a genuine successor of Pascal meeting the requirements of system engineering, yet also to satisfy my teacher's urge to present a systematic, consistent, appealing, and teachable framework for professional programming.

In later years, I was often asked whether indeed I had designed Pascal and Modula-2 as languages for teaching. The answer is "Yes, but not only". I wanted to teach *programming* rather than a language. A language, however, is needed to express programs. Thus, the language must be an appropriate tool, both for formulating programs and for expressing the basic concepts. It must be supportive, rather than a burden! But I also hasten to add that Pascal and Modula-2 were not intended to remain confined to the academic classroom. They were expected to be useful in practice. Further comments can be found in [44].

To be accurate, I had designed and implemented the predecessor language Modula [7 - 9] in 1975. It had been

conceived not as a general-purpose language, but rather as a small, custom-tailored language for experimenting with concurrent processes and primitives for their synchronization. Its features were essentially confined to this topic, such as process, signal, and monitor [6]. The monitor, representing critical regions with mutual exclusion, mutated into modules in Modula-2. Modula-2 was planned to be a full-scale language for implementing the entire software for the planned personal workstation *Lilith* [12, 19]. This had to include device drivers and storage allocator, as well as applications like document preparation and e-mail systems.

As it turned out later, Modula-2 was rather too complex. The result of an effort at simplification ten years later was Oberon.

2. The Language Modula-2

The defining report of Modula-2 appeared in 1979, and a textbook on it in 1982 [13]. A tutorial was published, following the growing popularity of the language [17, 18]. In planning Modula-2, I saw it as a new version of Pascal updated to the requirements of the time, and I seized the opportunity to correct various mistakes in Pascal's design, such as, for example, the syntactic anomaly of the dangling "else", the incomplete specification of procedure parameters, and others. Apart from relatively minor corrections and additions, the primary innovation was that of modules.

2.1 Modules

ALGOL had introduced the important notions of limited scopes of identifiers and of the temporary existence of objects. The limited visibility of an identifier and the limited lifetime of an object (variable, procedure), however, were tightly coupled: All existing objects were visible, and the ones that were not visible did not exist (were not allocated). This tight coupling was an obstacle in some situations. We refer to the well-known function to generate the next pseudo-random number, where the last one must be stored to compute the next, while remaining invisible. ALGOL's designers noticed this and quickly remedied the shortcoming by introducing the *own* property, an unlucky idea. An example is the following procedure for generating pseudo-random numbers (c_1 , c_2 , c_3 stand for constants):

```
real procedure random;
begin own real x;
    x := (c1*x + c2) mod c3; random := x
end
```

Here x is invisible outside the procedure. However, its computed value is retained and available the next time the procedure is called. Hence x cannot be allocated on a stack like ordinary local variables. The inadequacy of the *own* concept becomes apparent if one considers how to give an initial value to x .

Modula's solution was found in a second facility for establishing a scope, the *module*. In the first facility, the procedure (*block* in ALGOL), locally declared objects are allocated (on a stack) when control reaches the procedure, and deallocated when the procedure terminates. With the second, the module, no allocation is associated; only visibility is affected. The module merely constitutes a wall around the local objects, through which only those objects are visible that are explicitly specified in an "export" or an "import" list. In other words, the wall makes every identifier declared within a module invisible outside, unless it occurs in the *export list*, and it makes every identifier declared in a surrounding module - possibly the universe - invisible inside,

unless it occurs in the module's *import list*. This definition makes sense if modules are considered as nestable, and it represents the concept of *information hiding* as first postulated by Parnas in 1972 [4].

Visibility being controlled by modules and existence by procedures, the example of the pseudo-random number generator now turns out as follows in Modula-2. Local variables of modules are allocated when the module is loaded, and remain allocated until the module is explicitly discarded:

```
module RandomNumbers;
export random;
var x: real;
procedure random(): real;
begin x := (c1*x +c2) mod c3; return x
end random;
begin x := c0 (*seed*)
end RandomNumbers
```

The notation for a module was chosen identical to that of a monitor proposed by Hoare in 1974 [6], but lacks connotation of mutual exclusion of concurrent processes (as it was in Modula-1 [7]).

Modula-2's module can also be regarded as a representation of the concept of *abstract data type* postulated by Liskov in 1974 [5]. A module representing an abstract type exports the type, typically a record structured type, and the set of procedures and functions applicable to it. The type's structure remains invisible and inaccessible from the outside of the module. Such a type is called *opaque*. This makes it possible to postulate module invariants. Probably the most popular example is the stack. (For simplicity, we refrain from providing the guards $s.n < N$ for *push* and $s.n > 0$ for *pop*).

```
module Stacks;
export Stack, push, pop, init;
type Stack = record n: integer; (*0 ≤ n < N*)
    a: array N of real
end;
procedure push(var s: Stack; x: real);
begin s.a[s.n] := x; inc(s.n) end push;
procedure pop(var s: Stack): real;
begin dec(s.n); return s.a[s.n] end pop;
procedure init(var s: Stack);
begin s.n := 0 end init
end Stacks
```

Here, for example, it would be desirable to parameterize the type definition with respect to the stack's size and element type (here N and *real*). The impossibility to parametrize shows the limitations of Modula-2's module construct. As an aside, we note that in object-oriented languages the concept of data type is merged with the module concept and is called a *class*. The fact remains that the two notions have different purposes, namely data structuring and information hiding, and they should not be confused, particularly in languages used for teaching programming.

The basic idea behind Mesa's module concept was also information hiding, as communicated by Geschke, Morris and Mitchell in various discussions [10, 11]. But its emphasis was on decomposing very large systems into relatively large components, called modules. Hence, Mesa's modules were not nestable, but formed separate units of programs. Clearly, the key issue was to *interface*, to connect such modules. However, it was enticing to

unify the concepts of information hiding, nested modules, monitors, abstract data types, and Mesa system units into a single construct. In order to consider a (global) module as a program, we simply need to imagine a universe into which global modules are exported and from which they are imported.

A slight distinction between inner, nested modules and global modules seemed nevertheless advisable from both the conceptual and implementation aspects. After all, global modules appear as the parts of a large system that are typically implemented by different people or teams. The key idea is that such teams design the interfaces of their parts together, and then can proceed with the implementations of the individual modules in relative isolation. To support this paradigm, Mesa's module texts were split in two parts; the *implementation part* corresponds to the conventional "program". The *definition part* is the summary of information about the exported objects, the module's *interface*, and hence replaces the export list.

If we reformulate the example of module *Stacks* under this aspect, its definition part is

```
definition Stacks;
  type Stack;
  procedure push(var s: Stack; x: real);
  procedure pop(var s: Stack): real;
  procedure init(var s: Stack);
end Stacks
```

This, in fact, is exactly the information a user (client) of module *Stacks* needs to know. He must not be aware of the actual representation of stacks, which implementers may change even without notifying clients. Twenty-five years ago, this watertight and efficiently implemented way of type and version consistency checking put Mesa and Modula-2 way ahead of their successors, including the popular Java and C++.

Modula-2 allowed two forms of specifying imports. In the simple form, the module's identifier is included in the import list:

```
import M
```

In this case all objects exported by *M* become visible. For example, identifiers *push* and *pop* declared in *Stacks* are denoted by *Stacks.push* and *Stacks.pop* respectively. When using the second form

```
from M import x, P
```

the unqualified identifiers *x* and *P* denote the respective objects declared in *M*. This second form became most frequently used, but in retrospect proved rather misguided. First, it could lead to clashes if the same identifier was exported by two different (imported) modules. And second, it was not immediately visible in a program text where an imported identifier was declared.

A further point perhaps worth mentioning in this connection is the handling of exported enumeration types. The desire to avoid long export lists led to the (exceptional) rule that the export of an enumeration type identifier implies the export of all constant identifiers of that type. As nice as this may sound to the abbreviation enthusiast, it also has negative consequences, again in the form of identifier clashes. This occurs if two enumeration types are imported that happen to have a common constant identifier. Furthermore, identifiers may now appear that are neither locally declared, nor qualified by a module name, nor

visible in an import list, an entirely undesirable situation in a structured language.

Whereas the notation for the module concept is a matter of language design, the paradigm of system development by teams influenced the implementation technique, the way modules are compiled and linked. Actually, the idea of compiling parts of a program, such as subroutines, independently was not new; it existed since the time of FORTRAN. However, strongly typed languages add a new aspect: Type compatibility of variables and operators must be guaranteed not only among statements within a module, but also, and in particular, *between* modules. Hence, the term *separate compilation* was used in contrast to *independent compilation* without consistency checks between modules. With the new technique the definition (interface) of a module is compiled first, thereby generating a *symbol file*. This file is inspected not only upon compilation of the module itself, but also each time a client module is compiled. The specification of a module name in the import list causes the compiler to load the respective symbol file, providing the necessary information about the imported objects. A most beneficial consequence is that the inter-module checking occurs at the time of compilation rather than each time a module is linked.

One might object that this method is too complicated, and that the same effect is achieved by simply providing the service module's definition (interface) in source form whenever a client is compiled. Indeed, this solution was adopted, for example in Turbo Pascal with its *include files*, and in virtually all successors up to Java. But it misses the whole point. In system development, modules undergo changes, and they grow. In short, new *versions* emerge. This bears the danger of linking a client with wrong, old versions of servers - with disastrous consequences. A linker must guarantee the correct versions are linked, namely the same as were referenced upon compilation. This is achieved by letting the compiler provide every symbol file and every object file with a *version key*, and to make compilers and linkers check the compatibility of versions by inspecting their keys. This idea went back to Mesa; it quickly proved to be an immense benefit and soon became indispensable.

2.2 Procedure Types

An uncontroversial, fairly straightforward, and most essential innovation was the procedure type, also adopted from Mesa. In a restricted form it had been present also in Pascal, even ALGOL, namely in the form of parametric procedures. Hence, the concept needed only to be generalized, i.e. made applicable to parameters and variables. In respect to Pascal (and ALGOL), the mistake of incomplete parameter specification was amended, making the use of procedure types type-safe. This is an apparently minor, but in reality most essential point, because a type-consistency checking system is worthless if it contains loopholes.

2.3 The Type CARDINAL

The 1970s were the time of the 16-bit minicomputers. Their word length offered an address range from 0 to $2^{16}-1$, and thus a memory size of 64K. Whereas around 1970, this was still considered adequate, it later became a severe limitation, as memories became larger and cheaper. Unfortunately, computer architects insisted on byte addressing, thus covering only 32K words.

In any case, address arithmetic required unsigned arithmetic. Consequently, signed as well as unsigned arithmetic was offered, which effectively differed only in the interpretation of the sign bit in comparisons and in multiplication and division. We therefore

introduced the type CARDINAL ($0 \dots 2^{16}-1$) to the set of basic data types, in addition to INTEGER ($-2^{15} \dots 2^{15}-1$). This necessity caused several surprises. As a subtle example, the following statement, using a CARDINAL variable x , became unacceptable.

```
x := N-1; while x ≥ 0 do S(x); x := x-1 end
```

The correct solution, avoiding a negative value of x , is of course

```
x := N; while x > 0 do x := x-1; S(x) end
```

Also, the definitions of division and remainder raised problems. Whereas mathematically correct definitions of quotient q and remainder r of integer division x by y satisfy the equation

$$qxy + r = x, \quad 0 \leq r < y$$

which yields, for example $(7 \text{ div } 3) = 2$ and $(-7 \text{ div } 3) = -3$, and thus corresponds to the definitions postulated in Modula, although most available computers provided a division, where $(-x) \text{ div } y = -(x \text{ div } y)$, that is, $(-7 \text{ div } 3) = -2$.

2.4 Low-level Facilities

Facilities that make it possible to express situations that do not properly fit into the set of abstractions constituting the language, but rather mirror properties of the computer are called *low-level facilities*. Although necessary at the time – for example, to program device drivers and storage managers – I believe that they were introduced too lightheartedly, in the naive assumption that programmers would use them only sparingly and as a last resort. In particular, the concept of *type transfer function* was a major mistake. It allows the type identifier T to be used in expressions as a function identifier: The value of $T(x)$ is equal to x , whereby x is interpreted as being of type T , i.e. x is *cast* into a T . This interpretation inherently depends on the underlying (binary) representation of data types. Therefore, every program making use of this facility is inherently implementation-dependent, a clear contradiction of the fundamental goal of high-level languages.

In the same category of easily misused features is the *variant record*, a feature inherited from Pascal (see [13, Chap. 20]). The real stumbling block is the variant without tag field. The tag field's value is supposed to indicate the structure currently assumed by the record. If a tag is missing, it is impossible to determine the current variant. It is exactly this lack that can be misused to access record fields with intentionally "wrong" types.

2.5 What Was Left Out

Hoare used to remark that a language is indeed defined by the features it includes, but more so even by those that it excludes. My own guideline was to omit features whose correct semantics and best form were still unknown. Therefore it is worth while mentioning what was left out.

Concurrency was a hot topic, and still is. There was no clear favorite way to express and control concurrency, and hence no set of language constructs that clearly offered themselves for inclusion. One basic concept was seen in concurrent processes synchronized by *signals* (or conditions) and involving critical regions of mutual exclusion in the form of monitors [6]. Yet, it was decided that only the very basic notion of coroutines would be included in Modula-2, and that higher abstractions should be programmed as modules based on coroutines. This decision was even more plausible because the primitives could well be

classified as low-level facilities, and their realization encapsulated in a module (see [13, Chaps. 30 and 31]).

We also abandoned the belief that *interrupt handling* should be treated by the same mechanism as programmed process switching. Interrupts are typically subject to specific real-time conditions. Real-time response is impaired beyond acceptable limits if interrupts are handled by very general, complicated switching and scheduling routines.

Exception handling was widely considered a must for any language suitable for system programming. The concept originated from the need to react in special ways to rare situations, such as arithmetic overflow, index values being beyond a declared range, access via nil-pointer, etc., generally conditions of "unforeseen" error. Then the concept was extended to let any condition be declarable as an exception requiring special treatment. What lent this trend some weight was the fact that the code handling the exception might lie in a procedure different from the one in which the exception occurred (was *raised*), or even in a different module. This precluded the programming of exception handling by conventional conditional statements. Nevertheless, we decided not to include exception handling (with the exception of the ultimate exception called *Halt*).

Modula-2 features pointers and thereby implies dynamic storage allocation. Allocation of a variable x^\dagger is expressed by the standard procedure *Allocate(x)*, typically taking storage from a pool area (*heap*). A return of storage to the pool is signaled by the standard procedure *Deallocate(x)*. This was known to be a highly unsatisfactory solution, because it allows a program to return records to free storage that are still reachable from other, valid pointer variables, and therefore constitutes a rich source of disastrous errors.

The alternative is to postulate a *global storage management* that retrieves unused storage automatically, that is, a *garbage collector*. We rejected this for several reasons.

1. I believed it was better for programmers to devise their own storage managers, thus obtaining the most *effective use* of storage for the case at hand. This was of great importance at the time, considering the small memory size, such as 64k bytes for the PDP-11, on which Modula-2 was first implemented.
2. Garbage collectors could activate themselves at *unpredictable* times and hence preclude dependable real-time performance, which was considered an important domain of applications of Modula-2.
3. Garbage collectors must rely on incorruptible metadata about all variables in use. Given the many loopholes for breaching the typing system, it was considered impossible to devise secure garbage collection with reasonable effort. The flexibility of the language had become its own impediment. Even today, providing a garbage collector with an unsafe language is a sure guarantee of occasional crashes.

3. Implementations

Although a preliminary technical memorandum stating certain goals and concepts of the new language was written in 1977, the effective language design took place in 1978-79. Concurrently, a compiler implementation project was launched. The available machinery was a single DEC PDP-11 with a 64K-byte store. The single-pass strategy of our Pascal compilers could not be adopted; a multipass approach was unavoidable in view of the small

memory. It had actually been the Mesa implementation at Xerox's Palo Alto Research Center (PARC) that had proved possible what I had believed impracticable, namely to build a complete compiler operating on a small computer. The first Modula-2 compiler, written by van Le in 1977, consisted of seven passes, each generating sequential output written onto the 2M-byte disk. This number was reduced to five passes in a second design by Ammann in 1979. The first pass, the scanner, generated a token string and a hash table of identifiers. The second pass (parser) performed syntax analysis, and the third pass handled type checking. Passes 4 and 5 were devoted to code generation. This compiler was operational in early 1979.

In the meantime, a new Modula-2 compiler was designed in 1979-80 by Geissmann and Jacobi with the PDP-11 compiler as a guide, but taking advantage of the features of the new Lilith computer. *Lilith* was designed by the author and Ohran along the guidelines of the Xerox *Alto* [12, 19, 21]. It was based on the excellent Am2901 bit-slice processor of AMD and was microprogrammed. The new Modula-2 compiler consisted of only four passes, code generation being simplified due to the new architecture. Development took place on the PDP-11. Concurrently, the operating system *Medos* was implemented by Knudsen, a system essentially following in the footsteps of batch systems with program load and execute commands entered from the keyboard. At the same time, display and window software was designed by Jacobi. It served as the basis of the first application programs, such as a text editor - mostly used for program editing - featuring the well-known techniques of multiple windows, a cursor/mouse interface, and pop-up menus.

By 1981, Modula-2 was in daily use and quickly proved its worth as a powerful, efficiently implemented language. In December 1980, a pilot series of 20 Liliths, manufactured in Utah under Ohran's supervision, had been delivered to ETH Zürich. Further software development proceeded with a more than 20-fold hardware power at our disposal. A genuine personal workstation environment had successfully been established.

During 1984, the author designed and implemented yet another compiler for Modula-2. It was felt that compilation could be handled more simply and more efficiently if full use were made of the now available larger store which, by today's measures, was still very small. Lilith's 64K-word memory and its high code density allowed the realization of a single-pass compiler. This resulted in a dramatic reduction in disk operations. Indeed, compilation time of the compiler itself was reduced from some four minutes to a scant 45 seconds.

The new, much more compact compiler retained the partitioning of tasks. Instead of each task constituting a pass with sequential input from and output to disk, it constituted a module with a procedural interface. Common data structures, such as the symbol table, were defined in a data-definition module imported by (almost) all other modules. These modules represented a scanner, a parser, a code generator, and a handler of symbol files. During all these reimplementations, the language remained practically unchanged. The only significant change was the deletion of the explicit export lists in the definition parts of modules. The list is redundant because all identifiers declared in a definition text are exported. The compilation of imports and exports constituted a remarkable challenge for the objective of economy of linking data and with the absence of automatic storage management [24].

Over the years, it became clear that designers of control and data acquisition systems found Modula-2 particularly attractive. This was due to the presence of both low-level facilities to control interfaces and of modules to encapsulate critical, device-specific

parts. A Modula-2 compiler was offered by two British companies, but Modula-2 never experienced the same success as Pascal and never became as widely known. The primary reason was probably that Pascal had been riding on the back of the micro-computer wave invading homes and schools, reaching a class of people not infected by earlier programming languages and habits. Modula-2, on the other hand, was perceived as merely an upgrade on Pascal, hardly worth the effort of a language transition. The author, however, naively believed that everyone familiar with Pascal would happily welcome the additions and improvements of Modula-2.

Nevertheless, numerous implementation efforts proceeded at various universities for various computers [15, 16, 23]. Significant industrial projects adopted Modula-2. Among them was the control system for a new line of the Paris Metro and the entire software for a new Russian satellite navigation system. User's groups were established and conferences held, with structured programming in general and Pascal and Modula-2 in particular, as their themes. A series of tri-annual Joint Modular Languages Conferences (JMLC) started in 1987 in Bled (Slovenia), followed by events in Loughborough (England, 1990), Ulm (Germany, 1994), Linz (Austria, 1997), Zürich (Switzerland, 2000), Klagenfurt (Austria, 2003) and Oxford (England, 2006). Unavoidably, suggestions for extensions began to appear in the literature [25]. Even a direct successor was designed, called Modula-3 [33], in cooperation between DEC's Systems Research Center (SRC) and Olivetti's Research Laboratory in Palo Alto. Both had adopted Modula-2 as their principal system implementation language.

Closer to home, the Modula/Lilith project had a significant impact on our own future research and teaching activities. With 20 personal workstations available in late 1980, featuring a genuine high-level language, a very fast compiler, an excellent text editor, a high-resolution display, a mouse, an Ethernet connecting the workstations, a laser printer, and a central file server, we had in 1981 the first modern computing environment outside America. This gave us the opportunity to develop modern software for future computers fully five years before the first such system became commercially available, the Apple Macintosh, which was a scaled-down version of the Alto of 10 years before. Of particular value were our projects in modern document preparation and font design [20, 22].

4. From Modula to Oberon

As my sabbatical year at Xerox in 1976/77 had inspired me to design the personal workstation *Lilith* in conjunction with Modula-2, my second stay in 1984/85 provided the necessary inspiration and motivation for the language and operating system Oberon [29, 30]. Xerox PARC's Cedar system for its extremely powerful Dorado computer was based on the windows concept developed also at PARC for Smalltalk.

The Cedar system [14] was – to this author's knowledge – the first operating system that featured a mode of operation completely different from the then-conventional batch processing mode. In a batch system, a permanent loop accepts command lines from a standard input source. Each command causes the loading, execution, and release of a program. Cedar, in contrast, allowed many programs to remain allocated at the same time. It did not imply (storage) release after execution. Switching the processor from one program to another was done through the invocation of a program's commands, typically represented by *buttons* or *icons* in *windows* (called *viewers*) belonging to the program. This scheme had become feasible through the advent of large main stores (up to several hundred kilobytes), high-

resolution displays (up to 1000 by 800 pixels), and powerful processors (with clock rate up to 25 Megahertz).

Because I was supposed to teach the main course on system software and operating system design after my return from sabbatical leave, my encounter with the novel Cedar experiment appeared as a lucky coincidence. But how could I possibly teach the subject in good conscience without truly understanding it? Thus the idea was born of gaining first-hand experience by designing such a modern operating system on my own, with Cedar as the primary source of ideas.

Fortunately, my colleague Jürg Gutknecht concurrently spent a sabbatical semester at PARC in the summer of 1985. We both were intrigued by this new style of working with a computer, but at the same time also appalled by the system's complexity and lack of a clear, conceptual basis. This lack was probably due to the merging of several innovative ideas, and partly also due to the pioneers' contagious enthusiasm encouraged by the apparently unbounded future reservoir of hardware resources.

But how could the two of us possibly undertake and successfully complete such a large task? Were we not victims of an exuberant overestimation of our capabilities, made possible only by a naïve ignorance of the subject? We felt the strong urge to try and risk failure. We believed that a turnaround from the world-wide trend to more and more unmanageable complexity was long overdue. We felt that the topic was worthy of academic pursuit, and that ultimately teachers, students, and practitioners of computing would benefit from it.

We both felt challenged to mold the new concepts embodied by Cedar into a scheme that was clearly defined and therefore easy to teach and understand. Concentration on the essentials, omission of "nice-to-have" features, and careful planning before coding were no well-meaning guidelines heard in a classroom, but an absolute necessity considering the size of the task. The influence of the Xerox Lab was thus – the reader will excuse some oversimplification – twofold: We were inspired by *what* could be done, and shown *how not* to do it. The essential, conceptual ingredients of our intentions are summarized as follows:

1. Clear separation of the notion of *program* into the two independent notions of (1) the *module* as the unit of compilable text and of code and data to be loaded into the store (and discarded from it), and (2) the *procedure* as the unit of action invoked by a *command*.
2. The elimination of the concept of command lines written from the keyboard into a special command viewer. Actions would now be invoked by mouse-button clicking (middle button = command button) on the *command name* appearing in *any* arbitrary text in *any* viewer. The form of a command name, $M.P$, P denoting the procedure and M the module of which P is a part, would allow a simple search in the lists of loaded modules and M 's commands.
3. The core of execution being a tight loop located at the bottom of the system. In this loop the common sources of input (keyboard, mouse, net) are continuously sampled. Any input forming a command causes the dispatch of control to the appropriate procedure (by an *upcall*) if needed after the prior loading of the entire module containing it (*load on demand*). Note that this scheme excludes the preemption of program execution at arbitrary points.

4. Storage retrieval by a single, global *garbage collector*.

This is possible only under the presence of a watertight, preferably static *type-checking* concept. Deallocation of entire modules (code, global variables) occurs only through commands explicitly issued by the user.

5. Postulation of a simple syntax for (command) texts, paired with an *input scanner* parsing this syntax.

These five items describe the essence of our transition from batch mode operation to a modern, interactive multiviewer operating environment, manifest by the transition from the Modula-2 to the Oberon world [30, 35]. The clearly postulated conceptual basis made it possible for two programmers (J. Gutknecht and me) alone to implement the entire system, including the compiler and text processing machinery, in our spare time during only two years (1987-89). The tiny size of this team had a major influence on the conceptual consistency and integrity of the resulting system, and certainly also on its economy.

We emphasize that the aspects mentioned concern the *system* rather than the *language* Oberon. A language is an abstraction, a formal notation; notions such as command line, tight control loop, and garbage collector do not and must not occur in a language definition because they concern the implementation only. Therefore, let us now turn our attention to the language proper. As for the system, our intention was also for the language to strive for conceptual economy, to simplify Modula-2 where possible. As a consequence, our strategy was first to decide what should be omitted from Modula-2, and thereafter to decide which additions were necessary.

5. The Language Oberon

The programming language Oberon was the result of a concentrated effort to increase the power of Modula-2 and simultaneously to reduce its complexity. Oberon is the last member of a family of "ALGOL-like" languages that started with ALGOL 60, followed by ALGOL-W, Pascal, Modula-2, and ended with Oberon [27, 28]. By "ALGOL-like" is meant the procedural paradigm, a rigorously defined syntax, traditional mathematical notation for expressions (without esoteric $++$, $==$, $/=$ symbols), block structure providing scopes of identifiers and the concept of locality, the availability of recursion for procedures and functions, and a strict, static data typing scheme.

The principal guideline was to concentrate on features that are basic and essential and to omit ephemeral issues. This was certainly sensible in view of the very limited manpower available. But it was also driven by the recognition of the cancerous growth of complexity in languages that had recently emerged, such as C, C++ and Ada, which appeared even less suitable for teaching than for engineering in industry. Even Modula-2 now appeared overly bulky, containing features that we had rarely used. Their elimination would not cause a sacrifice. To try to crystallize the essential - not only the convenient and conventional - features into a small language seemed like a worthwhile (academic) exercise [28, 43].

5.1 Features Omitted from Oberon

A large number of standard data types not only complicates compilers but also makes it more difficult to teach and master a language. Hence, data types were a primary target of our simplification zeal.

An undisputed candidate for elimination was Modula's *variant record*. Introduced with the laudable intent of providing flexibility in data structuring, it ended up mostly being misused to

breach the typing concept. The feature allows interpretation of a data record in various ways according to various overlaid field templates, where one of them is marked as valid by the current value of a *tag field*. The true sin was that this tag could be omitted. For more details, the reader is referred to [13, Chap. 20].

Enumeration types would appear to be an attractive and innocent enough concept to be retained. However, a problem appeared in connection with import and export: Would the export of an enumeration type automatically also export the constants' identifiers, which would have to be prefixed with the module's name? Or, as in Modula-2, could these constant identifiers be used unqualified? The first option was unattractive because it produced unduly long names for constants, and the second because identifiers would appear without any declaration. As a consequence, the enumeration feature was dropped.

Subrange types were also eliminated. Experience had shown that they were used almost exclusively for indexing arrays. Hence, range checks were necessary for indexing rather than for assignment to a variable of subrange type. Lower array bounds were fixed to 0, making index checks more efficient and subrange types even less useful.

Set types had proved to be of limited usefulness in Pascal and Modula-2. Sets implemented as bit strings of the length of a "word" were rarely used, even though union and intersection could be computed by a single logical operation. In Oberon, we replaced general set types by the single, predefined type **set**, with elements 0 – 31.

After lengthy discussion, it was decided (in 1988) to merge the definition text of a module with its implementation text. This may have been a mistake from the pedagogical point of view. The definitions should clearly be designed first as contracts between its designer and the module's clients. Instead, now all exported identifiers were simply to be marked in their declaration (by an asterisk). The advantages of this solution were that a separate definition text became superfluous and that the compiler was relieved of consistency checking (of procedure signatures) between the two texts. An influential argument for the merger was that a separate definition text could be generated automatically from the module text.

The *qualified import* option of Modula-2 was dropped. Now every occurrence of an imported identifier must be preceded by its defining module's name. This actually turned out to be of great benefit when reading programs. The import list now contains module names only. This we believe to be a good example of the art of simplification: A simplified version of Mesa's module machinery was further simplified without compromising the essential ideas behind the facility: information hiding and type-safe separate compilation.

The number of *low-level facilities* was sharply reduced, and in particular type-transfer functions were eliminated. The few remaining low-level functions were encapsulated in a pseudo-module whose name would appear in the prominently visible import list of every module making use of such low-level facilities.

By eliminating all potentially unsafe facilities, the most essential step was finally made to a truly high-level language. Watertight type checking, also across modules, strict index checking at runtime, nil-pointer checking, and the safe type-extension concept let the programmer rely on the language rules alone. There was no longer a need to know about the underlying computer or how the language is translated and data are represented. The old goal, that a language must be defined without mentioning an executing mechanism, had finally been reached. Clean abstraction from machines and genuine portability

had become a reality. Apart from this, absolute type safety is (an often ignored truth) also an undisputable prerequisite for an underlying automatic storage management (garbage collector).

One feature must be mentioned that in hindsight should have been added: finalization, implying the automatic execution of a specified routine when a module is unloaded or a record (object) is collected. Inclusion of finalization had been discussed, but its cost and implementation effort had been judged too high relative to its benefit. However, its importance was underestimated, particularly that of a module being unloaded.

5.2 New Features Introduced in Oberon

Only two new features were introduced in Oberon: Type extension and type inclusion. This is surprising, considering the large number of eliminations.

The concept of *type inclusion* binds all arithmetic types together. Every one of them defines a range of values that variables of that type can assume. Oberon features five arithmetic types:

longreal \supseteq real \supseteq longint \supseteq integer \supseteq shortint

The concept implies that values of the included type can be assigned to variables of the including type. Hence, given

```
var i: integer; k: longint; x: real
```

assignments $k := i$ and $x := i$ are legal, whereas $i := k$ and $k := x$ are not. In hindsight, the fairly large number of arithmetic types looks like a mistake. The two types *integer* and *real* might have been sufficient. The decision was taken in view of the high importance of storage economy at the time, and because the target processor featured instruction sets for all five types. Of course, the language definition did not forbid implementations to treat *integer*, *longint*, and *shortint*, or *real* and *longreal* as the same.

The vastly more important new feature was *type extension* [26, 39]. Together with procedure-typed fields of record variables, it constitutes the technical foundation of the object-oriented programming style. The concept is better known under the anthropomorphic term *inheritance*. Consider a record type (class) *Window* (T_0) with coordinates x , y , width w and height h . It would be declared as

```
T0 = record x, y, w, h: integer end
```

T_0 may serve as the basis of an *extension* (subclass) *TextWindow* (T_1), declared as

```
T1 = record (T0) t: Text end
```

implying that T_1 retains (inherits) all properties, (x , y , w and h) from its *base type* T_0 , and in addition features a *text* field t . It also implies that all T_1 s are also T_0 s, thereby letting us to form heterogeneous data structures. For example, the elements of a tree may be defined as of type T_0 . However, individually assigned elements may be of any type that is an extension of T_0 , such as a T_1 .

The only new operation required is the *type test*. Given a variable v of type T_0 , the Boolean expression $v \text{ is } T$ is used to determine the effective, current type assigned to v . This is a runtime test.

Type extension alone, in addition to procedure types, is necessary for programming in object-oriented style. An *object*

type (class) is declared as a record containing procedure-typed fields, also called *methods*. For example:

```
type viewer = pointer to record x, y, w, h: integer;
    move: procedure (v: viewer; dx, dy: integer);
    draw: procedure (v: viewer; mode: integer);
end
```

The operation of drawing a certain viewer *v* is then expressed by the call *v.draw(v, 0)*. The first *v* serves to qualify the method *draw* as belonging to the type *viewer*, the second *v* designates the specific object to be drawn.

This reveals that object-oriented programming is effectively a style based on (inheriting) conventional procedural programming. Surprisingly, most people did not see Oberon as supporting object-orientation, simply because it did not use the new terminology. In 1990, Mössenböck spearheaded an effort to amend this apparent shortcoming and to implement a slight extension called Oberon-2 [34]. In Oberon-2 methods, i.e. procedures bound to a record type, were clearly marked as belonging to a record, and they implied a special parameter designating the object to which the method was to be applied. As a consequence, such methods were considered *constants* and therefore required an additional feature for replacing methods called *overriding*.

6. Implementations

The first ideas leading to Oberon were drafted in 1985, and the language was fully defined in early 1986 in close cooperation with J. Gutknecht. The report was only 16 pages long [28].

The first compiler was programmed by this author, deriving it from the single-pass Modula-2 compiler. It was written in (a subset of) Modula-2 for Lilith with the clear intention to translate it into Oberon, and it generated code for our Ceres workstation, equipped with the first commercial 32-bit microprocessor NS32032 of *National Semiconductor Corporation*. The compiled code was downloaded over a 2400 bit/s serial data line. As expected, the compiler was considerably simpler than its Modula-2 counterpart, although code generation for the NS processor was more complex than for Lilith's bytecode.

With the porting of the compiler completed, the development of the operating environment could begin. This system, (regrettably) also called Oberon, consisted of a file system, a display management system for windows (called *viewers*), a text system using multiple fonts, and backup to diskettes [30]. The entire system was programmed by Gutknecht and the author as a spare time activity over more than two years as described in [31]. The system was released in 1989, and then a larger number of developers began to generate applications. These included a network based on a low-cost RS-485 connection operating at 230 Kb/s [32], a laser printer, color displays (black and white was still the standard at the time), a laser printer, a mail and a file server, and more sophisticated document and graphics editors.

With the availability of a large number of Ceres workstations, Oberon was introduced in 1990 as the language for introductory courses at ETH Zürich [35, 36, 42], and also for courses in system software and compiler design. Having ourselves designed and implemented a complete system down to the last details, we were in a good position to teach software design. For many years, it had been our goal to publish a textbook not only sketching abstract principles, but also showing concrete examples. Our efforts resulted in a single book containing the complete source text of this compact yet real, useful, and convenient system. It was published in 1992 [37].

Following Hoare's earlier suggestion to write texts describing master sample programs to be studied and followed by students, we had published a text on widely useful algorithms and data structures, and now extended the idea to an entire operating system. Hoare had claimed that every other branch of engineering is taught by its underlying theoretical framework and by textbooks featuring concrete practical examples. However, interest in our demanding text remained disappointingly small. This may be explained in part by the custom in computer science of learning to write programs before reading any. Consequently, literature containing programs worth reading is rather scarce. A second reason for the low interest was that languages and operating systems were no longer popular topics of research. Also, among leading educational institutions the widespread feeling prevailed that the current commercial systems and languages were the end of the topic and here to stay. Their enormous size was taken as evidence that there was no chance for small research groups to contribute; arguing was considered as providing an irrelevant alternative with no chance of practical acceptance.

Nevertheless, we believe that the Oberon project was worth the effort and that its educational aspect was considerable. It may still serve as an example of how to strive for simplicity and perspicuity in complex situations. Gigantic commercial systems are highly inappropriate for studying principles and their economical realization. However, Oberon should not be considered merely "a teaching language". While it is suitable for teaching because it allows starting with a subset without mentioning the rest, it is powerful enough for large engineering projects.

During the years 1990 – 1995, Oberon received much attention, not the least because of our efforts to port it to the majority of commercial platforms. Compilers (code generators) were developed for the Intel xx86, the Motorola 680x0 (M. Franz), the Sun Sparc (J. Templ), the MIPS (R. Crelier) and the IBM Power (M. Brandis) processors [40]. The remarkable result of this concerted effort was that Oberon became a *truly portable platform*, allowing programs developed on one processor to compile and run on any other processor without adaptation.

Let us conclude this report with a peculiar story. In 1994 the author wrote yet another code generator, not for a different processor but rather for the same NS32000. This may seem strange and needs further explanation.

The NS processor had been chosen for Ceres because of its HLL-oriented instruction set, like that of Lilith. Among other features, it contained a large number of addressing modes, among which was the *external mode*. It corresponded to what was needed to address variables and call procedures of imported modules, and it allowed a fast linking process through the use of link tables. This sophisticated scheme made it possible to quickly load and link modules without any modification of the code. Only a simple link table had to be constructed, again similar to the case of Lilith.

The implementers of Oberon for other platforms had no such feature available. Nevertheless, they managed to find an acceptable solution. At the end, it turned out less complicated than feared, and I started to wonder how an analogous scheme used in the National Semiconductor processor would perform. To find out, I wrote a code generator using regular branch instructions (BSR) in place of the sophisticated external calls (CXP) and developed a linking loader adapted to the new scheme.

The new linker turned out to be not much more complicated, and hardly any slower. But execution of the new programs was considerably faster (up to 50%). This totally unexpected factor is explained by the development of the NS processor over various

versions and many years. In place of the 32032 in 1985, we used in 1988 the 32532 and in 1990 the 32GX32, which had the same instruction set, but were internally very different. The new versions were internally organized rather like RISC architectures, with the effect that simple, frequent instructions would execute very fast, while complex, rarely used instructions, such as our external calls, would perform poorly. Simple operations had become extremely fast (due to rising clock rates), whereas memory accesses remained relatively slow. On the other hand, memory capacity had grown tremendously. The relative importance of speed and code size had been changed. Hence, the old goal of high code density had become almost irrelevant.

The same phenomenon caused us to abandon the use of other "high-level" instructions, such as index bound checks and multiply-adds for computing matrix indices. This is a striking example of how hardware technology can very profoundly influence software design considerations.

7. Conclusions and Reflections

My long-term goal had been to demonstrate that a systematic design using a supportive language leads to lean, efficient, and economical software, requiring a fraction of the resources that is usually claimed. This goal has been reached successfully. I firmly believe, from many experiences over many years, that a structured language is instrumental in achieving a structured design. In addition, it was demonstrated that the clean, compact design of an entire software system can be described and explained in a single book [37]. The entire Oberon system, including its compiler, text editor and window system, occupied less than 200K bytes of main memory, and compiled itself in less than 40 seconds on a computer with a clock frequency of 25 MHz.

In the current year, however, such figures seem to have little significance. When the capacity of main memory is measured in hundreds of megabytes and disk space is available in dozens of gigabytes, 200K bytes do not count. When clock frequencies are of the order of gigahertz, the speed of compilation is irrelevant. Or, expressed the other way round, for a computer user to recognize a process as being slow, the software must be lousy indeed. The incredible advances in hardware technology have exerted a profound influence on software development. Whereas they allowed systems to reach phenomenal performance, their influence on the discipline of programming have been as a whole rather detrimental. They have permitted software quality and performance to drop disastrously, because poor performance is easily hidden behind faster hardware. In teaching, the notions of economizing memory space and processor cycles have become a thing apart. In fact, programming is hardly considered as a serious topic; it can be learnt by osmosis or, worse, by relying on extant program "libraries".

This stands in stark contrast to the times of ALGOL and FORTRAN. Languages were to be precisely defined, their unambiguity to be proven; they were to be the foundation of a logical, consistent framework for proving programs correct, not merely syntactically well-formed. Such an ambitious goal can be reached only if the framework is sufficiently small and simple. By contrast, modern languages are constantly growing. Their size and complexity is simply beyond anything that might serve as a logical foundation. In fact, they elude human grasp. Manuals have reached dimensions that effectively discourage any search for enlightenment. As a consequence, programming is not learnt from rules and logical derivations, but rather by trial and error. The glory of interactivity helps.

The world at large seems to tolerate this development. Given the staggering hardware power, one can usually afford to be wasteful of space and time. The boundaries will be hit in other dimensions: usability and reliability. The enormous size of current commercial systems limits understanding and fosters mistakes, leading to product unreliability. Signs that the limits of tolerance are being reached have begun to appear. Over the past few years I heard of a growing number of companies that had adopted Oberon as their exclusive programming tool. Their common characteristic was the small size and a small number of trusting and faithful clients requesting software of high quality, reliability and ease of use. Creating such software requires that its designers understand their products thoroughly. Naturally, this understanding must include the underlying operating system and the libraries on which the designs rest and rely. But the perpetual complexification of commercial software has made such understanding impossible. These companies have found Oberon the viable alternative.

Not surprisingly, these companies consist of small teams of expert programmers having the competence to make courageous decisions and enjoying the trust and confidence of a limited group of satisfied customers. It is not surprising that small systems like Oberon are finding acceptance primarily in the field of embedded systems for data acquisition and real-time control. Here, not only is economy a foremost concern, but even more so are reliability and robustness [45].

Still, these clients and applications are the exception. The market favors languages of commercial origin, regardless of their technical merits or defects. The market's inertia is enormous, as it is driven by a multitude of vicious circles that reinforce themselves. Hence, the value and role of creating new programming languages in research is a legitimate question that must be posed.

New ideas for improving the discipline of programming stem from practice. They are to be expressed in a notation, eventually forming a concrete language that is to be implemented and tested in the field. Insights thus gained find their way into new versions of widely used commercial languages slowly, very slowly over decades. It is fair to claim that Pascal, Modula-2, and Oberon have been successful in making such contributions over time.

The most essential of their messages, however, is expressed in the heading of the Oberon Report: "Make it as simple as possible, but not simpler". This advice has not yet been widely understood [41, 46]. It seems that currently commercial interests point in another direction. Time will tell.

Acknowledgments

The design of a language is the result of personal experiences in programming and of one's technical implementation knowhow, but also of discussions with and advice from others, and last but not least of struggles in teaching the art of programming. I am therefore very grateful to all who contributed either directly or indirectly, whether through support, praise, or criticism. My sincere thanks go to my former colleagues and coauthors J. Gutknecht and H. Mössenböck, and to the teams who implemented Modula-2 and Oberon and ported them to a variety of machine architectures. In particular, I am grateful to L. Geissmann, Ch. Jacobi, M. Franz, R. Crelier, M. Brandis, R. Griesemer, J. Templ, and C.Pfister. Without their untiring efforts and enthusiasm these languages would never have become so widely known and used. I express my thanks also to the referees of this article, in particular Greg Nelson for his insightful suggestions for improvement. And finally, I gratefully acknowledge the role of ETH (Federal Institute of Technology) in

funding and providing an environment of freedom from immediate needs and commercial pressures.

References

1. P. Naur and B. Randell, *Software Engineering*, Report of a conference sponsored by the NATO Science Committee. Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division, NATO (1969), p 231.
2. N. Wirth, "The Programming Language Pascal," *Acta Informatica*, 1 (June 1971), pp. 35-63.
3. O.-J. Dahl, E. W. Dijkstra, and C.A.R. Hoare, *Structured Programming*, Academic Press, 1972.
4. D. L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules". *Comm. ACM*, 15 (Dec. 1972), pp. 1053 – 1058.
5. B. Liskov and S. Zilles, "Programming with Abstract Datatypes", *Proc. ACM Conf. on Very High Level Languages, SIGPLAN Notices* 9, 4 (April 1974), pp. 50-59.
6. C.A.R. Hoare, "Monitors: An Operating System Structuring Concept", *Comm. ACM* 17, 10 (Oct. 1974), pp. 549 – 557.
7. N. Wirth, "Modula: A Language for Modular Multiprogramming," *Software - Practice and Experience*, 7 (1977), pp. 3-35.
8. --, "The Use of Modula," *Software - Practice and Experience*, 7 (1977), pp. 37-65.
9. --, "Design and Implementation of Modula," *Software - Practice and Experience*, 7 (1977), pp. 67 - 84.
10. C. Geschke, J. Morris, and E. Satterthwaite, "Early Experience with Mesa," *Comm. ACM* 20, 8, August 1977.
11. J. Mitchell, W. Maybury, and R. Sweet, "Mesa Language Manual," Xerox PARC Technical Report CSL-79-3, April 1979.
12. N. Wirth, "Lilith: A Personal Computer for the Software Engineer", *Proc. 5th Int'l Conf. Software Engineering*, San Diego, 1981, IEEE 81CH1627-9.
13. --, "Programming in Modula-2.", Springer-Verlag, Heidelberg, New York, 1982.
14. W. Teitelman, "A Tour through Cedar", *IEEE Software*, 1, 2 (April 1984), pp. 44-73.
15. T. L. Andersen, "Seven Modula Compilers Reviewed". *J. of Pascal, Ada, and Modula-2*, March-April 1984.
16. M. L. Powell, "A Portable, Optimizing Compiler for Modula-2", *SIGPLAN Notices* 19 (6) (June 1984), pp. 310 – 318.
17. N. Wirth, "History and Goals of Modula-2", *BYTE* (Aug. 1984), pp. 145-152.
18. J. Gutknecht, "Tutorial on Modula-2", *BYTE* Aug. 1984, pp. 157-176.
19. R. Ohran, "Lilith and Modula-2", *BYTE* (Aug. 1984), pp. 181-192.
20. J. Gutknecht and W. Winiger, "Andra: The Document Preparation System of the Personal Workstation Lilith". *Software - Practice & Experience*, 14 (1984), pp. 73-100.
21. G. Pomberger, *Lilith and Modula-2*. Hanser Verlag, 1985.
22. J. Gutknecht, "Concepts of the Text Editor Lara", *Comm. ACM*, 28, 9 (Sept. 1985), pp. 942-960.
23. P. H. Hartel and D. Starreveld, "Modula-2 Implementation Overview". *J. of Pascal, Ada, and Modula-2*, (July/Aug. 1985), pp. 9-23.
24. J. Gutknecht, "Separate Compilation in Modula-2", *IEEE Software*, (Nov. 1986), pp. 29-38.
25. P. Rovner, "Extending Modula-2 to Build Large, Integrated Systems". *IEEE Software*, (Nov. 1986), pp. 46-57.
26. N. Wirth, "Type Extension". *ACM TOPLAS*, 10, 2 (April 1988), pp. 204-214.
27. --, "From Modula to Oberon". *Software - Practice and Experience*, 18, 7 (July 1988), pp. 661-670.
28. --, "The Programming Language Oberon", *Software - Practice and Experience*, 18, 7 (July 1988), pp. 671-690.
29. --, "Oberon: A System for Workstations", *Microprocessing and Microprogramming* 24 (1988), pp. 3-8.
30. N. Wirth and J. Gutknecht, "The Oberon System." *Software - Practice and Experience*, 19, 9 (Sept. 1989), pp. 857-893.
31. N. Wirth, "Designing a System from Scratch." *Structured Programming*, 10, 1 (Jan. 1989), pp. 10-18.
32. --, "Ceres-Net: A Low-Cost Computer Network", *Software - Practice and Experience*, 20, 1 (Jan. 1990), pp. 13-24.
33. Greg Nelson, ed., *Systems Programming with Modula-3*, Prentice Hall 1991.
34. H. Mössenböck and N. Wirth, "The Programming Language Oberon-2", *Structured Programming*, 12 (1991), pp. 179-195.
35. M. Reiser, *The Oberon System*, Addison-Wesley, 1991.
36. M. Reiser and N. Wirth, *Programming in Oberon: Steps Beyond Pascal and Modula*, Addison-Wesley, 1992.
37. N. Wirth and J. Gutknecht, *Project Oberon*, Addison-Wesley, 1992.
38. N. Wirth. "Recollections about the development of Pascal," in *History of Programming Languages-II*, Thomas J. Bergin, Jr. and Richard G. Gibson, Jr. (eds), ACM Press and Addison-Wesley Publishing Company (1993), pp. 97-120.
39. H. Mössenböck, "Extensibility in the Oberon System", *Nordic Journal of Computing* 1 (1994), pp. 77 – 93.
40. M. Brandis, R. Crelier, M. Franz and J. Templ, "The Oberon System Family". *Software - Practice and Experience*, 25 (Dec. 1995), pp. 1331-1366.
41. N. Wirth, "A Plea for Lean Software", *IEEE Computer* (Feb. 1995), pp. 64-68.
42. A. Fischer and H. Marais, *The Oberon Companion*. 1998.
43. M. Franz, "Oberon – The Overlooked Jewel", in *The School of Niklaus Wirth*. L. Böszörmenyi, J. Gutknecht, G. Pomberger, eds. d-punkt.verlag, Heidelberg, 2000.
44. N. Wirth, "Pascal and Its Successors", in M. Broy and E. Denert, eds. *Software Pioneers*, Springer-Verlag, 2002, pp. 109 – 119.
45. A. Koltashev, "A Practical Approach to Software Portability Based on Strong Typing and Architectural Stratification", L. Böszörmenyi and P. Schojer, eds.: JMLC 2003, LNCS 2789, pp. 98-101, Springer-Verlag, 2003.
46. J. Maeda, *The Laws of Simplicity*, The MIT Press, 2006.

Evolving a language in and for the real world: C++ 1991-2006

Bjarne Stroustrup

Texas A&M University

www.research.att.com/~bs

Abstract

This paper outlines the history of the C++ programming language from the early days of its ISO standardization (1991), through the 1998 ISO standard, to the later stages of the C++0x revision of that standard (2006). The emphasis is on the ideals, constraints, programming techniques, and people that shaped the language, rather than the minutiae of language features. Among the major themes are the emergence of generic programming and the STL (the C++ standard library's algorithms and containers). Specific topics include separate compilation of templates, exception handling, and support for embedded systems programming. During most of the period covered here, C++ was a mature language with millions of users. Consequently, this paper discusses various uses of C++ and the technical and commercial pressures that provided the background for its continuing evolution.

Categories and Subject Descriptors K.2 [*History of Computing*]: systems

General Terms Design, Programming Language, History

Keywords C++, language use, evolution, libraries, standardization, ISO, STL, multi-paradigm programming

1. Introduction

In October 1991, the estimated number of C++ users was 400,000 [121]. The corresponding number in October 2004 was 3,270,000 [61]. Somewhere in the early '90s C++ left its initial decade of exponential growth and settled into a decade of steady growth. The key efforts over that time were to

1. use the language (obviously)
2. provide better compilers, tools, and libraries
3. keep the language from fragmenting into dialects
4. keep the language and its community from stagnating

Permission to make digital/hard copy of part of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date of appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Permission may be requested from the Publications Dept., ACM, Inc., 2 Penn Plaza, New York, NY 11201-0701, USA, fax:+1(212) 869-0481, permissions@acm.org

©2007 ACM 978-1-59593-766-7/2007/06-ART4 \$5.00

DOI 10.1145/1238844.1238848

<http://doi.acm.org/10.1145/1238844.1238848>

Obviously, the C++ community spent the most time and money on the first of those items. The ISO C++ standards committee tends to focus on the second with some concern for the third. My main effort was on the third and the fourth. The ISO committee is the focus for people who aim to improve the C++ language and standard library. Through such change they (we) hope to improve the state of the art in real-world C++ programming. The work of the C++ standards committee is the primary focus of the evolution of C++ and of this paper.

Thinking of C++ as a platform for applications, many have wondered why — after its initial success — C++ didn't shed its C heritage to "move up the food chain" and become a "truly object-oriented" applications programming language with a "complete" standard library. Any tendencies in that direction were squelched by a dedication to systems programming, C compatibility, and compatibility with early versions of C++ in most of the community associated with the standards committee. Also, there were never sufficient resources for massive projects in that community. Another important factor was the vigorous commercial opportunism by the major software and hardware vendors who each saw support for application building as *their* opportunity to distinguish themselves from their competition and to lock in their users. Minor players saw things in much the same light as they aimed to grow and prosper. However, there was no lack of support for the committee's activities within its chosen domain. Vendors rely on C++ for their systems and for highly demanding applications. Therefore, they have provided steady and most valuable support for the standards effort in the form of hosting and — more importantly — of key technical attending.

For good and bad, ISO C++ [66] remains a general-purpose programming language with a bias towards systems programming that

- is a better C
- supports data abstraction
- supports object-oriented programming
- supports generic programming

An explanation of the first three items from a historical perspective can be found in [120, 121]; the explanation of

“supports generic programming” is a major theme of this paper. Bringing aspects of generic programming into the mainstream is most likely C++’s greatest contribution to the software development community during this period.

The paper is organized in loose chronological order:

§1 Introduction

§2 Background: C++ 1979-1991 — early history, design criteria, language features.

§3 The C++ world in 1991 — the C++ standards process, chronology.

§4 Standard library facilities 1991-1998 — the C++ standard library with a heavy emphasis on its most important and innovative component: the STL (containers, algorithms, and iterators).

§5 Language features 1991-1998 — focusing on separate compilation of templates, exception safety, run-time type information and namespaces.

§6 Standards maintenance 1997-2003 — for stability, no additions were made to the C++ standard. However, the committee wasn’t idle.

§7 C++ in real-world use — application areas; applications programming vs. systems programming; programming styles; libraries, Application Binary Interfaces (ABIs), and environments; tools and research; Java, C#, and C; dialects.

§8 C++0x — aims, constraints, language features, and library facilities.

§9 Retrospective — influences and impact; beyond C++.

The emphasis is on the early and later years: The early years shaped current C++ (C++98). The later ones reflect the response to experience with C++98 as represented by C++0x. It is impossible to discuss how to express ideas in code without code examples. Consequently, code examples are used to illustrate key ideas, techniques, and language facilities. The examples are explained to make them accessible to non-C++ programmers. However, the focus of the presentation is the people, ideas, ideals, techniques, and constraints that shape C++ rather than on language-technical details. For a description of what ISO C++ is today, see [66, 126]. The emphasis is on straightforward questions: What happened? When did it happen? Who were there? What were their reasons? What were the implications of what was done (or not done)?

C++ is a living language. The main aim of this paper is to describe its evolution. However, it is also a language with a great emphasis on backwards compatibility. The code that I describe in this paper still compiles and runs today. Consequently, I tend to use the present tense to describe it. Given the emphasis on compatibility, the code will probably also run as described 15 years from now. Thus, my use of the present tense emphasizes an important point about the evolution of C++.

2. Background: C++ 1979-1991

The early history of C++ (up until 1991) is covered by my HOPL-II paper [120]. The standard reference for the first 15 years of C++ is my book *The Design and Evolution of C++*, usually referred to as D&E [121]. It tells the story from the pre-history of C++ until 1994. However, to set the scene for the next years of C++, here is a brief summary of C++’s early history.

C++ was designed to provide Simula’s facilities for program organization together with C’s efficiency and flexibility for systems programming. It was intended to deliver that to real projects within half a year of the idea. It succeeded.

At the time, I realized neither the modesty nor the preposterousness of that goal. The goal was modest in that it did not involve innovation, and preposterous in both its time scale and its Draconian demands on efficiency and flexibility. While a modest amount of innovation did emerge over the early years, efficiency and flexibility have been maintained without compromise. While the goals for C++ have been refined, elaborated, and made more explicit over the years, C++ as used today directly reflects its original aims.

Starting in 1979, while in the Computer Science Research Center of Bell Labs, I first designed a dialect of C called “C with Classes”. The work and experience with C with Classes from 1979 to 1983 determined the shape of C++. In turn, C with Classes was based on my experiences using BCPL and Simula as part of my PhD studies (in distributed systems) in the University of Cambridge, England. For challenging systems programming tasks I felt the need for a “tool” with the following properties:

- A good tool would have Simula’s support for program organization – that is, classes, some form of class hierarchies, some form of support for concurrency, and compile-time checking of a type system based on classes. This I saw as support for the process of inventing programs; that is, as support for design rather than just support for implementation.
- A good tool would produce programs that ran as fast as BCPL programs and share BCPL’s ability to combine separately compiled units into a program. A simple linkage convention is essential for combining units written in languages such as C, Algol68, Fortran, BCPL, assembler, etc., into a single program and thus not to suffer the handicap of inherent limitations in a single language.
- A good tool should allow highly portable implementations. My experience was that the “good” implementation I needed would typically not be available until “next year” and only on a machine I couldn’t afford. This implied that a tool must have multiple sources of implementations (no monopoly would be sufficiently responsive to users of “unusual” machines and to poor graduate students), that there should be no complicated run-time support system to port, and that there should be only very

limited integration between the tool and its host operating system.

During my early years at Bell Labs, these ideals grew into a set of “rules of thumb” for the design of C++.

- General rules:

- C++’s evolution must be driven by real problems.
- Don’t get involved in a sterile quest for perfection.
- C++ must be useful *now*.
- Every feature must have a reasonably obvious implementation.
- Always provide a transition path.
- C++ is a language, not a complete system.
- Provide comprehensive support for each supported style.
- Don’t try to force people to use a specific programming style.

- Design support rules:

- Support sound design notions.
- Provide facilities for program organization.
- Say what you mean.
- All features must be affordable.
- It is more important to allow a useful feature than to prevent every misuse.
- Support composition of software from separately developed parts.

- Language-technical rules:

- No implicit violations of the static type system.
- Provide as good support for user-defined types as for built-in types.
- Locality is good.
- Avoid order dependencies.
- If in doubt, pick the variant of a feature that is easiest to teach.
- Syntax matters (often in perverse ways).
- Preprocessor usage should be eliminated.

- Low-level programming support rules:

- Use traditional (dumb) linkers.
- No gratuitous incompatibilities with C.
- Leave no room for a lower-level language below C++ (except assembler).
- What you don’t use, you don’t pay for (zero-overhead rule).
- If in doubt, provide means for manual control.

These criteria are explored in detail in Chapter 4 of D&E [121]. C++ as defined at the time of release 2.0 in 1989 strictly fulfilled these criteria; the fundamental tensions in the effort to design templates and exception-handling mechanisms for C++ arose from the need to depart from some aspects of these criteria. I think the most important property of these criteria is that they are only loosely connected with specific programming language features. Rather, they specify constraints on solutions to design problems.

Reviewing this list in 2006, I’m struck by two design criteria (ideals) that are not explicitly stated:

- There is a direct mapping of C++ language constructs to hardware
- The standard library is specified and implemented in C++

Coming from a C background and being deep in the development of the C++98 standard (§3.1), these points (at the time, and earlier) seemed so obvious that I often failed to emphasize them. Other languages, such as Lisp, Smalltalk, Python, Ruby, Java, and C#, do not share these ideals. Most languages that provide abstraction mechanisms still have to provide the most useful data structures, such as strings, lists, trees, associative arrays, vectors, matrices, and sets, as built-in facilities, relying on other languages (such as assembler, C, and C++) for their implementation. Of major languages, only C++ provides general, flexible, extensible, and efficient containers implemented in the language itself. To a large extent, these ideals came from C. C has those two properties, but not the abstraction mechanisms needed to define nontrivial new types.

The C++ ideal – from day one of C with Classes (§2.1) – was that the language should allow optimal implementation of arbitrary data structures and operations on them. This constrained the design of abstraction mechanisms in many useful ways [121] and led to new and interesting implementation techniques (for example, see §4.1). In turn, this ideal would have been unattainable without the “direct mapping of C++ language constructs to hardware” criterion. The key idea (from C) was that the operators directly reflected hardware operations (arithmetic and logical) and that access to memory was what the hardware directly offered (pointers and arrays). The combination of these two ideals is also what makes C++ effective for embedded systems programming [131] (§6.1).

2.1 The Birth of C with Classes

The work on what eventually became C++ started with an attempt to analyze the UNIX kernel to determine to what extent it could be distributed over a network of computers connected by a local area network. This work started in April of 1979 in the Computing Science Research Center of Bell Laboratories in Murray Hill, New Jersey. Two subproblems soon emerged: how to analyze the network traffic that would result from the kernel distribution and how to modularize the

kernel. Both required a way to express the module structure of a complex system and the communication pattern of the modules. This was exactly the kind of problem that I had become determined never to attack again without proper tools. Consequently, I set about developing a proper tool according to the criteria I had formed in Cambridge.

In October of 1979, I had the initial version of a pre-processor, called Cpre, that added Simula-like classes to C. By March of 1980, this pre-processor had been refined to the point where it supported one real project and several experiments. My records show the pre-processor in use on 16 systems by then. The first key C++ library, called “the task system”, supported a coroutine style of programming [108, 115]. It was crucial to the usefulness of “C with Classes,” as the language accepted by the pre-processor was called, in most early projects.

During the April to October period the transition from thinking about a “tool” to thinking about a “language” had occurred, but C with Classes was still thought of primarily as an extension to C for expressing modularity and concurrency. A crucial decision had been made, though. Even though support of concurrency and Simula-style simulations was a primary aim of C with Classes, the language contained no primitives for expressing concurrency; rather, a combination of inheritance (class hierarchies) and the ability to define class member functions with special meanings recognized by the pre-processor was used to write the library that supported the desired styles of concurrency. Please note that “styles” is plural. I considered it crucial — as I still do — that more than one notion of concurrency should be expressible in the language.

Thus, the language provided general mechanisms for organizing programs rather than support for specific application areas. This was what made C with Classes and later C++ a general-purpose language rather than a C variant with extensions to support specialized applications. Later, the choice between providing support for specialized applications or general abstraction mechanisms has come up repeatedly. Each time, the decision has been to improve the abstraction mechanisms.

2.2 Feature Overview

The earliest features included classes, derived classes, public/private access control, type checking and implicit conversion of function arguments. In 1981, inline functions, default arguments, and the overloading of the assignment operator were added based on perceived need.

Since a pre-processor was used for the implementation of C with Classes, only new features, that is features not present in C, needed to be described and the full power of C was directly available to users. Both of these aspects were appreciated at the time. In particular, having C as a subset dramatically reduced the support and documentation work needed. C with Classes was still seen as a dialect of C. Furthermore, classes were referred to as “An Abstract Data

Type Facility for the C Language” [108]. Support for object-oriented programming was not claimed until the provision of virtual functions in C++ in 1983 [110].

A common question about “C with Classes” and later about C++ was “Why use C? Why didn’t you build on, say, Pascal?” One version of my answer can be found in [114]:

C is clearly not the cleanest language ever designed nor the easiest to use, so why do so many people use it?

- C is *flexible*: It is possible to apply C to most every application area, and to use most every programming technique with C. The language has no inherent limitations that preclude particular kinds of programs from being written.
- C is *efficient*: The semantics of C are ‘low level’; that is, the fundamental concepts of C mirror the fundamental concepts of a traditional computer. Consequently, it is relatively easy for a compiler and/or a programmer to efficiently utilize hardware resources for a C program.
- C is *available*: Given a computer, whether the tiniest micro or the largest super-computer, the chance is that there is an acceptable-quality C compiler available and that that C compiler supports an acceptably complete and standard C language and library. There are also libraries and support tools available, so that a programmer rarely needs to design a new system from scratch.
- C is *portable*: A C program is not automatically portable from one machine (and operating system) to another nor is such a port necessarily easy to do. It is, however, usually possible and the level of difficulty is such that porting even major pieces of software with inherent machine dependences is typically technically and economically feasible.

Compared with these ‘first-order’ advantages, the ‘second-order’ drawbacks like the curious C declarator syntax and the lack of safety of some language constructs become less important.

Pascal was considered a toy language [78], so it seemed easier and safer to add type checking to C than to add the features considered necessary for systems programming to Pascal. At the time, I had a positive dread of making mistakes of the sort where the designer, out of misguided paternalism or plain ignorance, makes the language unusable for real work in important areas. The ten years that followed clearly showed that choosing C as a base left me in the mainstream of systems programming where I intended to be. The cost in language complexity has been considerable, but (just barely) manageable. The problem of maintaining compatibility with an evolving C language and standard library is a serious one to this day (see §7.6).

In addition to C and Simula, I considered Modula-2, Ada, Smalltalk, Mesa, and Clu as sources for ideas for C++ [111], so there was no shortage of inspiration.

2.3 Work Environment

C with Classes was designed and implemented by me as a research project in the Computing Science Research Center of Bell Labs. This center provided a possibly unique environment for such work. When I joined in 1979, I was basically told to “do something interesting,” given suitable computer resources, encouraged to talk to interesting and competent people, and given a year before having to formally present my work for evaluation.

There was a cultural bias against “grand projects” requiring many people, against “grand plans” like untested paper designs for others to implement, and against a class distinction between designers and implementers. If you liked such things, Bell Labs and other organizations had many places where you could indulge such preferences. However, in the Computing Science Research Center it was almost a requirement that you — if you were not into theory — personally implemented something embodying your ideas and found users who could benefit from what you built. The environment was very supportive for such work and the Labs provided a large pool of people with ideas and problems to challenge and test anything built. Thus I could write in [114]: “There never was a C++ paper design; design, documentation, and implementation went on simultaneously. Naturally, the C++ front-end is written in C++. There never was a “C++ project” either, or a “C++ design committee”. Throughout, C++ evolved, and continues to evolve, to cope with problems encountered by users, and through discussions between the author and his friends and colleagues”.

2.4 From C with Classes to C++

During 1982, it became clear to me that C with Classes was a “medium success” and would remain so until it died. The success of C with Classes was, I think, a simple consequence of meeting its design aim: C with Classes helped organize a large class of programs significantly better than C. Crucially, this was achieved without the loss of run-time efficiency and without requiring unacceptable cultural changes in development organizations. The factors limiting its success were partly the limited set of new facilities offered over C and partly the pre-processor technology used to implement C with Classes. C with Classes simply didn’t provide support for people who were willing to invest significant effort to reap matching benefits: C with Classes was an important step in the right direction, but only one small step. As a result of this analysis, I began designing a cleaned-up and extended successor to C with Classes and implementing it using traditional compiler technology.

In the move from C with Classes to C++, the type checking was generally improved in ways that are possible only using a proper compiler front-end with full understanding of

all syntax and semantics. This addressed a major problem with C with Classes. In addition, virtual functions, function name and operator overloading, references, constants (`const`), and many minor facilities were added. To many, virtual functions were the major addition, as they enable object-oriented programming. I had been unable to convince my colleagues of their utility, but saw them as essential for the support of a key programming style (“paradigm”).

After a couple of years of use, release 2.0 was a major release providing a significantly expanded set of features, such as type-safe linkage, abstract classes, and multiple inheritance. Most of these extensions and refinements represented experience gained with C++ and could not have been added earlier without more foresight than I possessed.

2.5 Chronology

The chronology of the early years can be summarized:

1979 Work on C with Classes starts; first C with Classes use

1983 1st C++ implementation in use

1984 C++ named

1985 Cfront Release 1.0 (first commercial release); *The C++ Programming Language* (TC++PL) [112]

1986 1st commercial Cfront PC port (Cfront 1.1, Glockenspiel)

1987 1st GNU C++ release

1988 1st Oregon Software C++ release; 1st Zortech C++ release;

1989 Cfront Release 2.0; *The Annotated C++ Reference Manual* [35]; ANSI C++ committee (J16) founded (Washington, D.C.)

1990 1st ANSI X3J16 technical meeting (Somerset, New Jersey); templates accepted (Seattle, WA); exceptions accepted (Palo Alto, CA); 1st Borland C++ release

1991 1st ISO WG21 meeting (Lund, Sweden); Cfront Release 3.0 (including templates); *The C++ Programming Language (2nd edition)* [118]

On average, the number of C++ users doubled every 7.5 months from 1 in October 1979 to 400,000 in October of 1991 [121]. It is of course very hard to count users, but during the early years I had contacts with everyone who shipped compilers, libraries, books, etc., so I’m pretty confident of these numbers. They are also consistent with later numbers from IDC [61].

3. The C++ World in 1991

In 1991, the second edition of my *The C++ Programming Language* [118] was published to complement the 1989 language definition *The Annotated C++ Reference Manual* (“the ARM”) [35]. Those two books set the standard for C++ implementation and to some extent for programming techniques for years to come. Thus, 1991 can be seen as the

end of C++'s preparations for entry into the mainstream. From then on, consolidation was a major issue. That year, there were five C++ compilers to choose from (AT&T, Borland, GNU, Oregon, and Zortech) and three more were to appear in 1992 (IBM, DEC, and Microsoft). The October 1991 AT&T release 3.0 of Cfront (my original C++ compiler [121]) was the first to support templates. The DEC and IBM compilers both implemented templates and exceptions, but Microsoft's did not, thus seriously setting back efforts to encourage programmers to use a more modern style. The effort to standardize C++, which had begun in 1989, was officially converted into an international effort under the auspices of ISO. However, this made no practical difference as even the organizational meeting in 1989 had large non-US participation. The main difference is that we refer to ISO C++ rather than ANSI C++. The C++ programmer — and would-be C++ programmer — could now choose from among about 100 books of greatly varying aim, scope, and quality.

Basically, 1991 was an ordinary, successful year for the C++ community. It didn't particularly stand out from the years before or after. The reason to start our story here is that my HOPL-II paper [120] left off in 1991.

3.1 The ISO C++ Standards Process

For the C++ community, the ISO standards process is central: The C++ community has no other formal center, no other forum for driving language evolution, no other organization that cares for the language itself and for the community as a whole. C++ has no owner corporation that determines a path for the language, finances its development, and provides marketing. Therefore, the C++ standards committee became the place where language and standard library evolution is seriously considered. To a large extent, the 1991–2006 evolution of C++ was determined by what could be done by the volunteer individuals in the committee and how far the dreaded “design by committee” could be avoided.

The American National Standards Institute committee for C++ (ANSI J16) was founded in 1989; it now works under the auspices of INCITS (InterNational Committee for Information Technology Standards, a US organization). In 1991, it became part of an international effort under the auspices of ISO. Several other countries (such as France, Japan, and the UK) have their own national committees that hold their own meetings (in person or electronically) and send representatives to the ANSI/ISO meetings. Up until the final vote on the C++98 standard [63] in 1997, the committee met three times a year for week-long meetings. Now, it meets twice a year, but depends far more on electronic communications in between meetings. The committee tries to alternate meetings between the North American continent and elsewhere, mostly Europe.

The members of the J16 committee are volunteers who have to pay (about \$800 a year) for the privilege of doing all the work. Consequently, most members represent companies that are willing to pay fees and travel expenses, but

there is always a small number of people who pay their own way. Each company participating has one vote, just like each individual, so a company cannot stuff the committee with employees. People who represent their nations in the ISO (WG21) and participate in the national C++ panels pay or not according to their national standards organization rules. The J16 convener runs the technical sessions and does formal votes. The voting procedures are very democratic. First come one or more “straw votes” in a working group as part of the process to improve the proposal and gain consensus. Then come the more formal J16 votes in full committee where only accredited members vote. The final (ISO) votes are done on a per-nation basis. The aim is for consensus, defined as a massive majority, so that the resulting standard is good enough for everybody, if not necessarily exactly what any one member would have preferred. For the 1997/1998 final standards ballot, the ANSI vote was 43-0 and the ISO vote 22-0. We really did reach consensus, and even unanimity. I'm told that such clear votes are unusual.

The meetings tend to be quite collegial, though obviously there can be very tense moments when critical and controversial decisions must be made. For example, the debate leading up to the export vote strained the committee (§5.2). The collegiality has been essential for C++. The committee is the only really open forum where implementers and users from different — and often vigorously competing — organizations can meet and exchange views. Without a committee with a dense web of professional, personal, and organizational connections, C++ would have broken into a mess of feuding dialects. The number of people attending a meeting varies between 40 and 120. Obviously, the attendance increases when the meeting is in a location with many C++ programmers (e.g., Silicon Valley) or something major is being decided (e.g., should we adopt the STL?). At the Santa Cruz meeting in 1996, I counted 105 people in the room at the same time.

Most technical work is done by individuals between meetings or in working groups. These working groups are officially “ad hoc” and have no formal standing. However, some lasts for many years and serve as a focus for work and as the institutional memory of the committee. The main long-lived working groups are:

- *Core* — chairs: Andrew Kornig, Jose Lajorie, Bill Gibbons, Mike Miller, Steve Adamczyk.
- *Evolution* (formerly *extensions*) — chair: Bjarne Stroustrup.
- *Library* — chairs: Mike Vilot, Beman Dawes, Matt Austern, Howard Hinnant.

When work is particularly hectic, these groups split into sub-working-groups focussed on specific topics. The aim is to increase the degree of parallelism in the process and to better use the skills of the many people present.

The official “chair” of the whole committee whose primary job is the ensure that all formal rules are obeyed and to report back to SC22 is called the convener. The original convener was Steve Carter (BellCore). Later Sam Harbinson (Tartan Labs and Texas Instruments), Tom Plum (Plum Hall), and Herb Sutter (Microsoft) served.

The J16 chairman conducts the meeting when the whole committee is gathered. Dmitri Lenkov (Hewlett-Packard) was the original J16 chair and Steve Clamage (Sun) took over after him.

The draft standard text is maintained by the project editor. The original project editor was Jonathan Shopiro (AT&T). He was followed by Andrew Koenig (AT&T) whose served 1992-2004, after which Pete Becker (Dinkumware) took over “the pen”.

The committee consists of individuals of diverse interests, concerns, and backgrounds. Some represent themselves, some represent giant corporations. Some use PCs, some use UNIX boxes, some use mainframes, etc. Some would like C++ to become more of an object-oriented language (according to a variety of definitions of “object-oriented”), others would have been more comfortable had ANSI C been the end point of C’s evolution. Many have a background in C, some do not. Some have a background in standards work, many do not. Some have a computer science background, some do not. Most are programmers, some are not. Some are language lawyers, some are not. Some serve end-users, some are tools suppliers. Some are interested in large projects, some are not. Some are interested in C compatibility, some are not. It is hard to find a generalization that covers them all.

This diversity of backgrounds has been good for C++; only a very diverse group could represent the diverse interests of the C++ community. The end results — such as the 1998 standard and the Technical Reports (§6.1, §6.2) — are something that is good enough for everyone represented, rather than something that is ideal for any one subcommunity. However, the diversity and size of the membership do make constructive discussion difficult and slow at times. In particular, this very open process is vulnerable to disruption by individuals whose technical or personal level of maturity doesn’t encourage them to understand or respect the views of others. Part of the consideration of a proposal is a process of education of the committee members. Some members have claimed — only partly in jest — that they attend to get that education. I also worry that the voice of C++ users (that is, programmers and designers of C++ applications) can be drowned by the voices of language lawyers, would-be language designers, standards bureaucrats, implementers, tool builders, etc.

To get an idea about what organizations are represented, here are some names from the 1991-2005 membership lists: Apple, AT&T, Bellcore, Borland, DEC, Dinkumware, Edison Design Group (EDG), Ericsson, Fujitsu, Hewlett-

Packard, IBM, Indiana University, Los Alamos National Labs, Mentor Graphics, Microsoft, NEC, Object Design, Plum Hall, Siemens Nixdorf, Silicon Graphics, Sun Microsystems, Texas Instruments, and Zortech.

Changing the definition of a widely used language is very different from simple design from first principles. Whenever we have a “good idea”, however major or minor, we must remember that

- there are hundreds of millions of lines of code “out there”
 - most will not be rewritten however much gain might result from a rewrite
- there are millions of programmers “out there” — most won’t take out time to learn something new unless they consider it essential
- there are decade-old compilers still in use — many programmers can’t use a language feature that doesn’t compile on every platform they support
- there are many millions of outdated textbooks out there
 - many will still be in use in five years’ time

The committee considers these factors and obviously that gives a somewhat conservative bias. Among other things, the members of the committee are indirectly responsible for well over 100 million lines of code (as representatives of their organizations). Many of the members are on the committee to promote change, but almost all do so with a great sense of caution and responsibility. Other members feel that their role is more along the lines of avoiding unnecessary and dangerous instability. Compatibility with previous versions of C++ (back to ARM C++ [35]), previous versions of C (back to K&R C [76]), and generations of corporate dialects is a serious issue for most members. We (the members of the committee) try to face future challenges, such as concurrency, but we do so remembering that C++ is at the base of many tool chains. Break C++ and the major implementations of Java and C# would also break. Obviously, the committee couldn’t “break C++” by incompatibilities even if it wanted to. The industry would simply ignore a seriously incompatible standard and probably also start migrating away from C++.

3.2 Chronology

Looking forward beyond 1991, we can get some idea of the process by listing some significant decisions (votes):

1993 Run-time type identification accepted (Portland, Oregon) §5.1.2; namespaces accepted (Munich, Germany) §5.1.1

1994 string (templatized by character type) (San Diego, California)

1994 The STL (San Diego, California) §4.1

1996 export (Stockholm, Sweden) §5.2

- 1997** Final committee vote on the complete standard (Morristown, New Jersey)
- 1998** ISO C++ standard [63] ratified
- 2003** Technical Corrigendum (“mid-term bug-fix release”) [66]; work on C++0x starts
- 2004** Performance technical report [67] §6.1; Library Technical Report (hash tables, regular expressions, smart pointers, etc.) [68] §6.2
- 2005** 1st votes on features for C++0x (Lillehammer, Norway); `auto`, `static_assert`, and rvalue references accepted in principle; §8.3.2
- 2006** 1st full committee (official) votes on features for C++0x (Berlin, Germany)

The city names reflect the location of the meeting where the decision was taken. They give a flavor of the participation. When the committee meets in a city, there are usually a dozen or more extra participants from nearby cities and countries. It is also common for the host to take advantage of the influx of C++ experts — many internationally known — to arrange talks to increase the understanding of C++ and its standard in the community. The first such arrangement was in Lund in 1991 when the Swedish delegation collaborated with Lund University to put on a two-day C++ symposium.

This list gives a hint of the main problem about the process from the point of view of someone trying to produce a coherent language and library, rather than a set of unrelated “neat features”. At any time, the work focused on a number of weakly related specific topics, such as the definition of “undefined”, whether it is possible to resume from an exception (it isn’t), what functions should be provided for `string`, etc. It is extremely hard to get the committee to agree on an overall direction.

3.3 Why Change?

Looking at the list of decisions and remembering the committee’s built-in conservative bias, the obvious question is: “Why change anything?” There are people who take the position that “standardization is to document existing practice”. They were never more than a tiny fraction of the committee membership and represent an even smaller proportion of the vocal members of the C++ committee. Even people who say that they want “no change” ask for “just one or two improvements”. In this, the C++ committee strongly resembles other language standardization groups.

Basically, we (the members of the committee) desire change because we hold the optimistic view that better language features and better libraries lead to better code. Here, “better” means something like “more maintainable”, “easier to read”, “catches more errors”, “faster”, “smaller”, “more portable”, etc. People’s criteria differ, sometimes drastically. This view is optimistic because there is ample evidence that people can — and do — write really poor code in every lan-

guage. However, most groups of programmers — including the C++ committee — are dominated by optimists. In particular, the conviction of a large majority of the C++ committee has consistently been that the quality of C++ code can be improved over the long haul by providing better language features and standard-library facilities. Doing so takes time: in some cases we may have to wait for a new generation of programmers to get educated. However, the committee is primarily driven by optimism and idealism — moderated by vast experience — rather than the cynical view of just giving people what they ask for or providing what “might sell”.

An alternative view is that the world changes and a living language will change — whatever any committee says. So, we can work on improvements — or let others do it for us. As the world changes, C++ must evolve to meet new challenges. The most obvious alternative would not be “death” but capture by a corporation, as happened with Pascal and Objective C, which became Borland and Apple corporate languages, respectively.

After the Lund (Sweden) meeting in 1991, the following cautionary tale became popular in the C++ community:

We often remind ourselves of the good ship Vasa. It was to be the pride of the Swedish navy and was built to be the biggest and most beautiful battleship ever. Unfortunately, to accommodate enough statues and guns, it underwent major redesigns and extension during construction. The result was that it only made it halfway across Stockholm harbor before a gust of wind blew it over and it sank, killing about 50 people. It has been raised and you can now see it in a museum in Stockholm. It is a beauty to behold — far more beautiful at the time than its unextended first design and far more beautiful today than if it had suffered the usual fate of a 17th century battleship — but that is no consolation to its designer, builders, and intended users.

This story is often recalled as a warning against adding features (that was the sense in which I told it to the committee). However, to complicate matters, there is another side to the story: Had the Vasa been completed as originally designed, it would have been sent to the bottom full of holes the first time it encountered a “modern two-deck” battleship. Ignoring changes in the world isn’t an option (either).

4. The Standard Library: 1991-1998

After 1991, most major changes to the draft C++ standard were in the standard library. Compared to that, the language features were little changed even though we made an apparently infinite number of improvements to the text of the standard. To gain a perspective, note that the standard is 718 pages: 310 define the language and 366 define the standard library. The rest are appendices, etc. In addition, the C++ standard library includes the C standard library by reference;

that's another 81 pages. To compare, the base document for the standardization, the reference manual of TC++PL2 [118], contained 154 pages on the language and just one on the standard library.

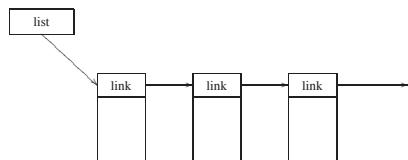
By far the most innovative component of the standard library is “the STL” — the containers, iterators, and algorithms part of the library. The STL influenced and continues to influence not only programming and design techniques but also the direction of new language features. Consequently, this is treated first here and in far greater detail than other standard library components.

4.1 The STL

The STL was the major innovation to become part of the standard and the starting point for much of the new thinking about programming techniques that have occurred since. Basically, the STL was a revolutionary departure from the way the C++ community had been thinking about containers and their use.

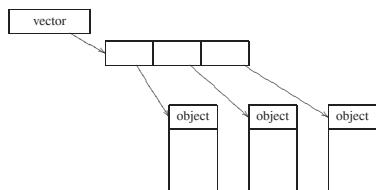
4.1.1 Pre-STL containers

From the earliest days of Simula, containers (such as lists) had been intrusive: An object could be put into a container if and only if its class had been (explicitly or implicitly) derived from a specific `Link` and/or `Object` class. This class contains the link information needed for management of objects in a container and provides a common type for elements. Basically, such a container is a container of references (pointers) to links/objects. We can graphically represent such a list like this:

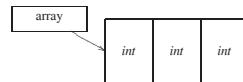


The links come from a base class `Link`.

Similarly, an “object-oriented” vector is a basically an array of references to objects. We can graphically represent such a vector like this:



The references to objects in the vector data structure point to objects of an `Object` base class. This implies that objects of a fundamental type, such as `int` and `double`, can't be put directly into containers and that the array type, which directly supports fundamental types, must be different from other containers:



Furthermore, objects of really simple classes, such as `complex` and `Point`, can't remain optimal in time and space if we want to put them into a container. In other words, Simula-style containers are intrusive, relying on data fields inserted into their element types, and provide indirect access to objects through pointers (references). Furthermore, Simula-style containers are not statically type safe. For example, a `Circle` may be added to a `list`, but when it is extracted we know only that it is an `Object` and need to apply a cast (explicit type conversion) to regain the static type.

Thus, Simula containers provide dissimilar treatment of built-in and user-defined types (only some of the latter can be in containers). Similarly, arrays are treated differently from user-defined containers (only arrays can hold fundamental types). This is in direct contrast to two of the clearest language-technical ideals for C++:

- Provide the same support for built-in and user-defined types
- What you don't use, you don't pay for (zero-overhead rule).

Smalltalk has the same fundamental approach to containers as Simula, though it makes the base class universal and thus implicit. The problems also appear later languages, such as Java and C# (though they – like Smalltalk – make use of a universal class and C# 2.0 applies C++-like specialization to optimize containers of integers). Many early C++ libraries (e.g. the NIHCL [50], early AT&T libraries [5]) also followed this model. It does have significant utility and many designers were familiar with it. However, I considered this double irregularity and the inefficiency (in time and space) that goes with it unacceptable for a truly general-purpose library (you can find a summary of my analysis in §16.2 of TC++PL3 [126]).

The lack of a solution to these logical and performance problems was the fundamental reason behind my “biggest mistake” of not providing a suitable standard library for C++ in 1985 (see D&E [121] §9.2.3): I didn't want to ship anything that couldn't directly handle built-in types as elements and wasn't statically type safe. Even the first “C with Classes” paper [108] struggled with that problem, unsuccessfully trying to solve it using macros. Furthermore, I specifically didn't want to provide something with covariant containers. Consider a `Vector` of some “universal” `Object` class:

```

void f(Vector& p)
{
    p[2] = new Pear;
}

void g()
  
```

```

{
    Vector apples(10); // container of apples
    for(int i=0; i<10; ++i)
        apples[i] = new Apple;
    f(apples);
    // now apples contains a pear
}

```

The author of g pretends that apples is a Vector of Apples. However, Vector is a perfectly ordinary object-oriented container, so it really contains (pointers to) Objects. Since Pear is also an Object, f can without any problems put a Pear into it. It takes a run-time check (implicit or explicit) to catch the error/misconception. I remember how shocked I was when this problem was first explained to me (sometime in the early 1980s). It was just too shoddy. I was determined that no container written by me should give users such nasty surprises and significant run-time checking costs. In modern (i.e. post-1988) C++ the problem is solved using a container parameterized by the element type. In particular, Standard C++ offers `vector<Apple>` as a solution. Note that a `vector<Apple>` does not convert to a `vector<Fruit>` even when Apple implicitly converts to Fruit.

4.1.2 The STL emerges

In late 1993, I became aware of a new approach to containers and their use that had been developed by Alex Stepanov. The library he was building was called “The STL”. Alex then worked at HP Labs but he had earlier worked for a couple of years at Bell Labs, where he had been close to Andrew Koenig and where I had discussed library design and template mechanisms with him. He had inspired me to work harder on generality and efficiency of some of the template mechanisms, but fortunately he failed to convince me to make templates more like Ada generics. Had he succeeded, he wouldn’t have been able to design and implement the STL!

Alex showed the latest development in his decades-long research into generic programming techniques aiming for “the most general and most efficient code” based on a rigorous mathematical foundation. It was a framework of containers and algorithms. He first explained his ideas to Andrew, who after playing with the STL for a couple of days showed it to me. My first reaction was puzzlement. I found the STL style of containers and container use very odd, even ugly and verbose. For example, you sort a vector of doubles, vd, according to their absolute value like this:

```
sort(vd.begin(), vd.end(), Absolute<double>());
```

Here, `vd.begin()` and `vd.end()` specify the beginning and end of the vector’s sequence of elements and `Absolute<double>()` compares absolute values.

The STL separates algorithms from storage in a way that’s classical (as in math) but not object oriented. Furthermore, it separates policy decisions of algorithms, such as sorting criteria and search criteria, from the algorithm, the

container, and the element class. The result is unsurpassed flexibility and — surprisingly — performance.

Like many programmers acquainted with object-oriented programming, I thought I knew roughly how code using containers had to look. I imagined something like Simula-style containers augmented by templates for static type safety and maybe abstract classes for iterator interfaces. The STL code looked very different. However, over the years I had developed a checklist of properties that I considered important for containers:

1. Individual containers are simple and efficient.
2. A container supplies its “natural” operations (e.g., list provides put and get and vector provides subscripting).
3. Simple operators, such as member access operations, do not require a function call for their implementation.
4. Common functionality can be provided (maybe through iterators or in a base class)
5. Containers are by default statically type-safe and homogeneous (that is, all elements in a container are of the same type).
6. A heterogeneous container can be provided as a homogeneous container of pointers to a common base.
7. Containers are non-intrusive (i.e., an object need not have a special base class or link field to be a member of a container).
8. A container can contain elements of built-in types
9. A container can contain structs with externally imposed layouts.
10. A container can be fitted into a general framework (of containers and operations on containers).
11. A container can be sorted without it having a `sort` member function.
12. “Common services” (such as persistence) can be provided in a single place for a set of containers (in a base class?).

A slightly differently phrased version can be found in [124].

To my amazement the STL met all but one of the criteria on that list! The missing criterion was the last. I had been thinking of using a common base class to provide services (such as persistence) for all derived classes (e.g., all objects or all containers). However, I didn’t (and don’t) consider such services intrinsic to the notion of a container. Interestingly, some “common services” can be expressed using “concepts” (§8.3.3) that specifically address the issue of what can be required of a set of types, so C++0x (§8) is likely to bring the STL containers even closer to the ideals expressed in that list.

It took me some time — weeks — to get comfortable with the STL. After that, I worried that it was too late to introduce a completely new style of library into the C++ community.

Considering the odds to get the standards committee to accept something new and revolutionary at such a late stage of the standards process, I decided (correctly) that those odds were very low. Even at best, the standard would be delayed by a year — and the C++ community urgently needed that standard. Also, the committee is fundamentally a conservative body and the STL was revolutionary.

So, the odds were poor, but I plodded on hoping. After all, I really did feel very bad about C++ not having a sufficiently large and sufficiently good standard library [120] (D&E [121] §9.2.3). Andrew Koenig did his best to build up my courage and Alex Stepanov lobbied Andy and me as best he knew how to. Fortunately, Alex didn't quite appreciate the difficulties of getting something major through the committee, so he was less daunted and worked on the technical aspects and on teaching Andrew and me. I began to explain the ideas behind the STL to others; for example, the examples in D&E §15.6.3.1 came from the STL and I quoted Alex Stepanov: “C++ is a powerful enough language — the first such language in our experience — to allow the construction of generic programming components that combine mathematical precision, beauty, and abstractness with the efficiency of non-generic hand-crafted code”. That quote is about generic programming in general (§7.2.1) and the STL in particular.

Andrew Koenig and I invited Alex to give an evening presentation at the October 1993 standards committee meeting in San Jose, California: “It was entitled *The Science of C++ Programming* and dealt mostly with axioms of regular types — connecting construction, assignment and equality. I also described axioms of what is now called Forward Iterators. I did not at all mention any containers and only one algorithm: *find*”. [105]. That talk was an audacious piece of rabble rousing that to my amazement and great pleasure basically swung the committee away from the attitude of “it's impossible to do something major at this stage” to “well, let's have a look”.

That was the break we needed! Over the next four months, we (Alex, his colleague Meng Lee, Andrew, and I) experimented, argued, lobbied, taught, programmed, redesigned, and documented so that Alex was able to present a complete description of the STL to the committee at the March 1994 meeting in San Diego, California. Alex arranged a meeting for C++ library implementers at HP later in 1994. The participants were Meng Lee (HP), Larry Podmolik, Tom Keffer (Rogue Wave), Nathan Myers, Mike Vilot, Alex, and I. We agreed on many principles and details, but the size of the STL emerged as the major obstacle. There was no consensus about the need for large parts of the STL, there was a (realistic) worry that the committee wouldn't have the time to properly review and more formally specify something that large, and people were simply daunted by the sheer number of things to understand, implement, document, teach, etc. Finally, at Alex's urging, I took a pen and literally

crossed out something like two thirds of all the text. For each facility, I challenged Alex and the other library experts to explain — very briefly — why it couldn't be cut and why it would benefit most C++ programmers. It was a horrendous exercise. Alex later claimed that it broke his heart. However, what emerged from that slashing is what is now known as the STL [103] and it made it into the ISO C++ standard at the October 1994 meeting in Waterloo, Canada — something that the original and complete STL would never have done. Even the necessary revisions of the “reduced STL” delayed the standard by more than a year. The worries about size and complexity (even after my cuts) were particularly acute among library implementers concerned about the cost of providing a quality implementation. For example, I remember Roland Hartinger (representing Siemens and Germany) worrying that acceptance of the STL would cost his department one million marks. In retrospect, I think that I did less damage than we had any right to hope for.

Among all the discussions about the possible adoption of the STL one memory stands out: Beman Dawes calmly explaining to the committee that he had thought the STL too complex for ordinary programmers, but as an exercise he had implemented about 10% of it himself so he no longer considered it beyond the standard. Beman was (and is) one of the all too rare application builders in the committee. Unfortunately, the committee tends to be dominated by compiler, library, and tools builders.

I credit Alex Stepanov with the STL. He worked with the fundamental ideals and techniques for well over a decade before the STL, unsuccessfully using languages such as Scheme and Ada [101]. However, Alex is always the first to insist that others took part in that quest. David Musser (a professor at Rensselaer Polytechnic Institute) has been working with Alex on generic programming for almost two decades and Meng Lee worked closely with him at HP helping to program the original STL. Email discussions between Alex and Andrew Koenig also helped. Apart from the slashing exercise, my technical contributions were minor. I suggested that various information related to memory be collected into a single object — what became the allocators. I also drew up the initial requirement tables on Alex's blackboard, thus creating the form in which the standard specifies the requirements that STL templates place on their arguments. These requirements tables are actually an indicator that the language is insufficiently expressive — such requirements should be part of the code; see §8.3.3.

Alex named his containers, iterators, and algorithm library “the STL”. Usually, that's considered an acronym for “Standard Template Library”. However, the library existed — with that name — long before it was any kind of standard and most parts of the standard library rely on templates. Wits have suggested “STepanov and Lee” as an alternative explanation, but let's give Alex the final word: “ ‘STL’ stands for

'STL' ". As in many other cases, an acronym has taken on a life of its own.

4.1.3 STL ideals and concepts

So what is the STL? It comes from an attempt to apply the mathematical ideal of generality to the problem of data and algorithms. Consider the problem of storing objects in containers and writing algorithms to manipulate such objects. Consider this problem in the light of the ideals of direct, independent, and composable representation of concepts:

- express concepts directly in code
- express relations among concepts directly in code
- express independent concepts in independent code
- compose code representing concepts freely wherever the composition makes sense

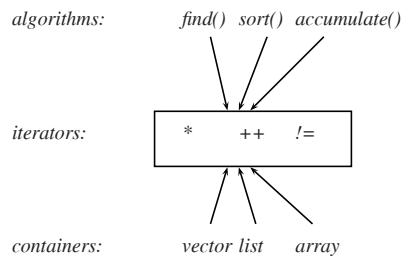
We want to be able to

- store objects of a variety of types (e.g. ints, Points, and pointers to Shapes)
- store those objects in a variety of containers (e.g. list, vector, and map),
- apply a variety of algorithms (e.g. sort, find, and accumulate) to the objects in the containers, and
- use a variety of criteria (comparisons, predicates, etc.) for those algorithms

Furthermore, we want the use of these objects, containers, and algorithms to be statically type safe, as fast as possible, as compact as possible, not verbose, and readable. Achieving all of this simultaneously is not easy. In fact, I spent more than ten years unsuccessfully looking for a solution to this puzzle (§4.1.2).

The STL solution is based on parameterizing containers with their element types and on completely separating the algorithms from the containers. Each type of container provides an iterator type and all access to the elements of the container can be done using only iterators of that type. The iterator defines the interface between the algorithm and the data on which it operates. That way, an algorithm can be written to use iterators without having to know about the container that supplied them. Each type of iterator is completely independent of all others except for supplying the same semantics to required operations, such as * and ++.

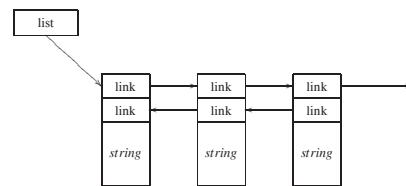
Algorithms use iterators and container implementers implement iterators for their containers:



Let's consider a fairly well-known example, the one that Alex Stepanov initially showed to the committee (San Jose, California, 1993). We want to find elements of various types in various containers. First, here are a couple of containers:

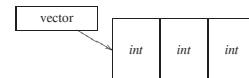
```
vector<int> vi; // vector of ints
list<string> ls; // list of strings
```

The `vector` and `list` are the standard library versions of the notions of `vector` and `list` implemented as templates. An STL container is a non-intrusive data structure into which you can copy elements of any type. We can graphically represent a (doubly linked) `list<string>` like this:



Note that the link information is *not* part of the element type. An STL container (here, `list`) manages the memory for its elements (here, `strings`) and supplies the link information.

Similarly, we can represent a `vector<int>` like this:

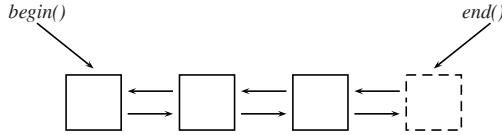


Note that the elements are stored in memory managed by the `vector` and `list`. This differs from the Simula-style containers in §4.1.1 in that it minimizes allocation operations, minimizes per-object memory, and saves an indirection on each access to an element. The corresponding cost is a copy operation when an object is first entered into a container; if a copy is expensive, programmers tend to use pointers as elements.

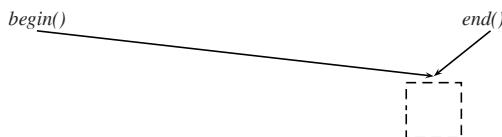
Assume that the containers `vi` and `ls` have been suitably initialized with values of their respective element types. It then makes sense to try to find the first element with the value 777 in `vi` and the first element with the value "Stepanov" in `ls`:

```
vector<int>::iterator p
= find(vi.begin(), vi.end(), 777);
list<string>::iterator q
= find(ls.begin(), ls.end(), "Stepanov");
```

The basic idea is that you can consider the elements of any container as a sequence of elements. A container “knows” where its first element is and where its last element is. We call an object that points to an element “an iterator”. We can then represent the elements of a container by a pair of iterators, `begin()` and `end()`, where `begin()` points to the first element and `end()` to one-beyond-the-last element. We can represent this general model graphically:



The `end()` iterator points to one-past-the-last element rather than to the last element to allow the empty sequence not to be a special case:



What can you do with an iterator? You can get the value of the element pointed to (using `*` just as with a pointer), make the iterator point to the next element (using `++` just as with a pointer) and compare two iterators to see if they point to the same element (using `==` or `!=` of course). Surprisingly, this is sufficient for implementing `find()`:

```
template<class Iter, class T>
Iter find(Iter first, Iter last, const T& val)
{
    while (first!=last && *first!=val)
        ++first;
    return first;
}
```

This is a simple — very simple, really — function template. People familiar with C and C++ pointers should find the code easy to read: `first!=last` checks whether we reached the end and `*first!=val` checks whether we found the value that we were looking for (`val`). If not, we increment the iterator `first` to make it point to the next element and try again. Thus, when `find()` returns, its value will point to either the first element with the value `val` or one-past-the-last element (`end()`). So we can write:

```
vector<int>::iterator p =
    find(vi.begin(), vi.end(), 7);

if (p != vi.end()) { // we found 7
    // ...
}
else { // no 7 in vi
    // ...
}
```

This is very, very simple. It is simple like the first couple of pages in a math book and simple enough to be really fast. However, I know that I wasn't the only person to take significant time figuring out what really is going on here and longer to figure out why this is actually a good idea. Like simple math, the first STL rules and principles generalize beyond belief.

Consider first the implementation: In the call `find(vi.begin(), vi.end(), 7)`, the iterators `vi.begin()` and `vi.end()` become `first` and `last`, respectively, inside. To `find()`, `first` is simply “something that points to an `int`”. The obvious implementation of `vector<int>::iterator` is therefore a pointer to `int`, an `int*`. With that implementation, `*` becomes pointer dereference, `++` becomes pointer increment, and `!=` becomes pointer comparison. That is, the implementation of `find()` is obvious and optimal.

Please note that the STL does not use function calls to access the operations (such as `*` and `!=`) that are effectively arguments to the algorithm because they depend on a template argument. In this, templates differ radically from most mechanisms for “generics”, relying on indirect function calls (like virtual functions), as provided by Java and C#. Given a good optimizer, `vector<int>::iterator` can without overhead be a class with `*` and `++` provided as inline functions. Such optimizers are now not uncommon and using a iterator class rather than a pointer improves type checking by catching unwarranted assumptions, such as that the iterator for a vector is a pointer:

```
int* p = find(vi.begin(), vi.end(), 7); // error

// verbose, but correct:
vector<int>::iterator q =
    find(vi.begin(), vi.end(), 7);
```

C++0x will provide ways of dealing with the verbosity; see §8.3.2.

In addition, *not* defining the interface between an algorithm and its type arguments as a set of functions with unique types provides a degree of flexibility that proved very important [130] (§8.3.3). For example, the standard library algorithm `copy` can copy between different container types:

```
void f(list<int>& lst, vector<int>& v)
{
    copy(lst.begin(), lst.end(), v.begin());
    // ...
```

```

    copy(v.begin(), v.end(), lst.end());
}

```

So why didn't we just dispense with all that "iterator stuff" and use pointers? One reason is that `vector<int>::iterator` could have been a class providing range checked access. For a less subtle explanation, have a look at another call of `find()`:

```

list<string>::iterator q =
    find(ls.begin(),ls.end(),"McIlroy");

if (q != ls.end()) { // we found "McIlroy"
    // ...
}
else { // no "McIlroy" in ls
    // ...
}

```

Here, `list<string>::iterator` isn't going to be a `string*`. In fact, assuming the most common implementation of a linked list, `list<string>::iterator` is going to be a `Link<string>*` where `Link` is a link node type, such as:

```

template<class T> struct Link {
    T value;
    Link* suc;
    Link* pre;
};

```

That implies that `*` means `p->value` ("return the value field"), `++` means `p->suc` ("return a pointer to the next link"), and `!=` pointer comparison (comparing `Link*`s). Again the implementation is obvious and optimal. However, it is completely different from what we saw for `vector<int>::iterator`.

We used a combination of templates and overload resolution to pick radically different, yet optimal, implementations of operations used in the definition of `find()` for each use of `find()`. Note that there is no run-time dispatch, no virtual function calls. In fact, there are only calls of trivially inlined functions and fundamental operations, such as `*` and `++` for a pointer. In terms of execution time and code size, we have hit the absolute minimum!

Why not use "sequence" or "container" as the fundamental notion rather than "pair of iterators"? Part of the reason is that "pair of iterators" is simply a more general concept than "container". For example, given iterators, we can sort the first half of a container only: `sort(vi.begin(), vi.begin() + vi.size() / 2)`. Another reason is that the STL follows the C++ design rules that we must provide transition paths and support built-in and user-defined types uniformly. What if someone kept data in an ordinary array? We can still use the STL algorithms. For example:

```

int buf[max];
// ... fill buf ...
int* p = find(buf,buf+max,7);

```

```

if (p != buf+max) { // we found 7
    // ...
}
else { // no 7 in buf
    // ...
}

```

Here, the `*`, `++`, and `!=` in `find()` really are pointer operations! Like C++ itself, the STL is compatible with older notions such as C arrays. Thus, the STL meets the C++ ideal of always providing a transition path (§2). It also meets the ideal of providing uniform treatment to user-defined types (such as `vector`) and built-in types (in this case, `array`) (§2).

Another reason for basing algorithms on iterators, rather than on containers or an explicit sequence abstraction, was the desire for optimal performance: using iterators directly rather than retrieving a pointer or an index from another abstraction eliminates a level of indirection.

As adopted as the containers and algorithms framework of the ISO C++ standard library, the STL consists of a dozen containers (such as `vector`, `list`, and `map`) and data structures (such as `arrays`) that can be used as sequences. In addition, there are about 60 algorithms (such as `find`, `sort`, `accumulate`, and `merge`). It would not be reasonable to present all of those here. For details, see [6, 126].

So, we can use simple arithmetic to see how the STL technique of separating algorithms from containers reduces the amount of source code we have to write and maintain. There are 60×12 (that is, 720) combinations of algorithm and container in the standard but just $60 + 12$ (that is, 72) definitions. The separation reduces the combinatorial explosion to a simple addition. If we consider element types and policy parameters (function objects, see §4.1.4) for algorithms we see an even more impressive gain: Assume that we have N algorithms with M alternative criteria (policies) and X containers with Y element types. Then, the STL approach gives us $N+M+X+Y$ definitions whereas "hand-crafted code" requires $N \times M \times X \times Y$ definitions. In real designs, the difference isn't quite that dramatic because typically designers attack that huge $N \times M \times X \times Y$ figure with a combination of conversions (one container to another, one data type to another), class derivations, function parameters, etc., but the STL approach is far cleaner and more systematic than earlier alternatives.

The key to both the elegance and the performance of the STL is that it — like C++ itself — is based directly on the hardware model of memory and computation. The STL notion of a sequence is basically that of the hardware's view of memory as a set of sequences of objects. The basic semantics of the STL map directly into hardware instructions allowing algorithms to be implemented optimally. The compile-time resolution of templates and the perfect inlining they support is then key to the efficient mapping of the high-level expression using the STL to the hardware level.

4.1.4 Function objects

The STL and generic programming in general owes a — freely and often acknowledged (e.g., [124]) — debt to functional programming. So where are the lambdas and higher-order functions? C++ doesn't directly support anything like that (though there are always proposals for nested functions, closures, lambdas, etc.; see §8.2). Instead, classes that define the application operator, called function objects (or even “functors”), take that role and have become the main mechanism of parameterization in modern C++. Function objects build on general C++ mechanisms to provide unprecedented flexibility and performance.

The STL framework, as described so far, is somewhat rigid. Each algorithm does exactly one thing in exactly the way the standard specifies it to. For example, using `find()`, we find an element that is equal to the value we give as the argument. It is actually more common to look for an element that has some desired property, such as matching strings without case sensitivity or matching floating-point values allowing for very slight differences.

As an example, instead of finding a value 7, let's look for a value that meets some predicate, say, being less than 7:

```
vector<int>::iterator p =
    find_if(v.begin(), v.end(), Less_than<int>(7));

if (p != vi.end()) { // element < 7
    // ...
}
else { // no such element
    // ...
}
```

What is `Less_than<int>(7)`? It is a function object; that is, it is an object of a class that has the application operator, `()`, defined to perform an action:

```
template<class T> struct Less_than {
    T value;
    Less_than(const T& v) :value(v) { }
    bool operator()(const T& v) const
        { return v<value; }
};
```

For example:

```
Less_than<double> f(3.14); // f holds 3.14
bool b1 = f(3); // true: 3<3.14 is true
bool b2 = f(4); // false: 4<3.14 is false
```

From the vantage point of 2005, it seems odd that function objects are not mentioned in D&E or TC++PL1. They deserve a whole section. Even the use of a user-defined application operator, `()`, isn't mentioned even though it has had a long and distinguished career. For example, it was among the initial set of operators (after `=`; see D&E §3.6) that I allowed to be overloaded and was among many other things used to mimic Fortran subscript notation [112].

We used the STL algorithm `find_if` to apply `Less_than<int>(7)` to the elements of the vector. The definition of `find_if` differs from `find()`'s definition in using a user-supplied predicate rather than equality:

```
template<class In, class Pred>
In find_if(In first, In last, Pred pred)
{
    while (first!=last && !pred(*first))
        ++first;
    return first;
}
```

We simply replaced `*first!=val` with `!pred(*first)`. The function template `find_if()` will accept any object that can be called given an element value as its argument. In particular, we could call `find_if()` with an ordinary function as its third argument:

```
bool less_than_7(int a)
{
    return a<7;
}

vector<int>::iterator p =
    find_if(v.begin(), v.end(), less_than_7);
```

However, this example shows why we often prefer a function object over a function: The function object can be initialized with one (or more) arguments and carry information along for later use. A function object can carry a state. That makes for more general and more elegant code. If needed, we can also examine that state later. For example:

```
template<class T>
struct Accumulator { // keep the sum of n values
    T value;
    int count;
    Accumulator() :value(0), count(0) { }
    Accumulator(const T& v) :value(v), count(0) { }
    void operator()(const T& v)
        { ++count; value+=v; }
};
```

An `Accumulator` object can be passed to an algorithm that calls it repeatedly. The partial result is carried along in the object. For example:

```
int main()
{
    vector<double> v;
    double d;
    while (cin>>d) v.push_back(d);

    Accumulator<double> ad;
    ad = for_each(v.begin(), v.end(), ad);
    cout << "sum==" << ad.value
        << ", mean==" << ad.value/ad.count
        << '\n';
    return 0;
}
```

The standard library algorithm `for_each` simply applies its third argument to each element of its sequence and returns that argument as its return value. The alternative to using a function object would be a messy use of global variables to hold value and count. In a multithreaded system, such use of global variables is not just messy, but gives incorrect results.

Interestingly, simple function objects tend to perform better than their function equivalents. The reason is that they tend to be simple classes without virtual functions, so that when we call a member function the compiler knows exactly which function we are calling. That way, even a simple-minded compiler has all the information needed to inline. On the other hand, a function used as a parameter is passed as a pointer, and optimizers have traditionally been incapable of performing optimizations involving pointers. This can be very significant (e.g. a factor of 50 in speed) when we pass an object or function that performs a really simple operation, such as the comparison criteria for a sort. In particular, inlining of function objects is the reason that the STL (C++ standard library) `sort()` outperforms the conventional `qsort()` by several factors when sorting arrays of types with simple comparison operators (such as `int` and `double`) [125]. Spurred on by the success of function objects, some compilers now can do inlining for pointers to functions as long as a constant is used in the call. For example, today, some compilers can inline calls to compare from within `qsort`:

```
bool compare(double* a, double* b) { /* ... */ }
// ...
qsort(p, max, sizeof(double), compare);
```

In 1994, no production C or C++ compiler could do that.

Function objects are the C++ mechanism for higher-order constructs. It is not the most elegant expression of high-order ideas, but it is surprisingly expressive and inherently efficient in the context of a general purpose language. To get the same efficiency of code (in time and space) from a general implementation of conventional functional programming facilities requires significant maturity from an optimizer. As an example of expressiveness, Jaakko Järvi and Gary Powell showed how to provide and use a lambda class that made the following example legal with its obvious meaning [72]:

```
list<int> lst;
// ...
Lambda x;
list<int>::iterator p =
    find_if(lst.begin(), lst.end(), x<7);
```

Note how overload resolution enables us to make the element type, `int`, implicit (deduced). If you want just `<` to work, rather than building a general library, you can add definitions for `Lambda` and `<` in less than a dozen lines of code. Using `Less_than` from the example above, we can simply write:

```
class Lambda {};

template<class T>
Less_than<T> operator<(Lambda, const T& v)
{
    return Less_than<T>(v);
}
```

So, the argument `x<7` in the call of `find_if` becomes a call of `operator<(Lambda, const int&)`, which generates a `Less_than<int>` object. That's exactly what we used explicitly in the first example in this section. The difference here is just that we have achieved a much simpler and more intuitive syntax. This is a good example of the expressive power of C++ and of how the interface to a library can be simpler than its implementation. Naturally, there is no run-time or space overhead compared to a laboriously written loop to look for an element with a value less than 7.

The closest that C++ comes to higher-order functions is a function template that returns a function object, such as `operator<` returning a `Less_than` of the appropriate type and value. Several libraries have expanded that idea into quite comprehensive support for functional programming (e.g., Boost's Function objects and higher-order programming libraries [16] and FC++ [99]).

4.1.5 Traits

C++ doesn't offer a general compile-time way of asking for properties of a type. In the STL, and in many other libraries using templates to provide generic type-safe facilities for diverse types, this became a problem. Initially, the STL used overloading to deal with this (e.g., note the way the type `int` is deduced and used in `x<7`; §4.1.4). However, that use of overloading was unsystematic and therefore unduly difficult and error prone. The basic solution was discovered by Nathan Myers during the effort to templatize `iostream` and `string` [88]. The basic idea is to provide an auxiliary template, "a trait", to contain the desired information about a set of types. Consider how to find the type of the elements pointed to by an iterator. For a `list_iterator<T>` it is `list_iterator<T>::value_type` and for an ordinary pointer `T*` it is `T`. We can express that like this:

```
template<class Iter>
struct iterator_trait {
    typedef Iter::value_type value_type;
};

template<class T>
struct iterator_trait<T*> {
    typedef T value_type;
};
```

That is, the `value_type` of an iterator is its member type `value_type`. However, pointers are a common form of iterators and they don't have any member types. So, for pointers we use the type pointed to as `value_type`. The language

construct involved is called partial specialization (added to C++ in 1995; §5). Traits can lead to somewhat bloated source code (though they have no object code or run-time cost) and even though the technique is very extensible it often requires explicit programmer attention when a new type is added. Traits are now ubiquitous in template-based C++ libraries. However, the “concepts” mechanism (§8.3.3) promises to make many traits redundant by providing direct language support for the idea of expressing properties of types.

4.1.6 Iterator categories

The result of the STL model as described so far could easily have become a mess with each algorithm depending on the peculiarities of the iterators offered by specific containers. To achieve interoperability, iterator interfaces have to be standardized. It would have been simplest to define a single set of operators for every iterator. However, to do so would have been doing violence to reality: From an algorithmic point of view lists, vectors, and output streams really do have different essential properties. For example, you can efficiently subscript a vector, you can add an element to a list without disturbing neighboring elements, and you can read from an input stream but not from an output stream. Consequently the STL provides a classification of iterators:

- input iterator (ideal for homogeneous stream input)
- output iterator (ideal for homogeneous stream output)
- forward iterator (we can read and write an element repeatedly, ideal for singly-linked lists)
- bidirectional iterator (ideal for doubly-linked lists)
- random access iterator (ideal for vectors and arrays)

This classification acts as a guide to programmers who care about interoperability of algorithms and containers. It allows us to minimize the coupling of algorithms and containers. Where different algorithms exist for different iterator categories, the most suitable algorithm is automatically chosen through overloading (at compile time).

4.1.7 Complexity requirements

The STL included complexity measures (using the big-O notation) for every standard library operation and algorithm. This was novel in the context of a foundation library for a language in major industrial use. The hope was and still is that this would set a precedent for better specification of libraries. Another — less innovative — aspect of this is a fairly systematic use of preconditions and postconditions in the specification of the library.

4.1.8 Stepanov’s view

The description of the STL here is (naturally) focused on language and library issues in the context of C++. To get a complementary view, I asked Alexander Stepanov for his perspective [106]:

In October of 1976 I observed that a certain algorithm — parallel reduction — was associated with monoids: collections of elements with an associative operation. That observation led me to believe that it is possible to associate every useful algorithm with a mathematical theory and that such association allows for both widest possible use and meaningful taxonomy. As mathematicians learned to lift theorems into their most general settings, so I wanted to lift algorithms and data structures. One seldom needs to know the exact type of data on which an algorithm works since most algorithms work on many similar types. In order to write an algorithm one needs only to know the properties of operations on data. I call a collection of types with similar properties on which an algorithm makes sense the *underlying concept* of the algorithm. Also, in order to pick an efficient algorithm one needs to know the complexity of these operations. In other words, complexity is an essential part of the interface to a concept.

In the late ’70s I became aware of John Backus’s work on FP [7]. While his idea of programming with functional forms struck me as essential, I realized that his attempt to permanently fix the number of functional forms was fundamentally wrong. The number of functional forms — or, as I call them now, generic algorithms — is always growing as we discover new algorithms. In 1980 together with Dave Musser and Deepak Kapur I started working on a language Tecton to describe algorithms defined on algebraic theories. The language was functional since I did not realize at the time that memory and pointers were a fundamental part of programming. I also spent time studying Aristotle and his successors which led me to better understanding of fundamental operations on objects like equality and copying and the relation between whole and part.

In 1984 I started collaborating with Aaron Kershbaum who was an expert on graph algorithms. He was able to convince me to take arrays seriously. I viewed sequences as recursively definable since it was commonly perceived to be the “theoretically sound” approach. Aaron showed me that many fundamental algorithms depended on random access. We produced a large set of components in Scheme and were able to implement generically some complicated graph algorithms.

The Scheme work led to a grant to produce a generic library in Ada. Dave Musser and I produced a generic library that dealt with linked structures. My attempts to implement algorithms that work on any sequential structure (both lists and arrays) failed because of the state of Ada compilers at the time. I had equivalences to many STL algorithms, but could not compile them.

Based on this work, Dave Musser and I published a paper where we introduced the notion of generic programming insisting on deriving abstraction from useful efficient algorithms. The most important thing I learned from Ada was the value of static typing as a design tool. Bjarne Stroustrup had learned the same lesson from Simula.

In 1987 at Bell Labs Andy Koenig taught me semantics of C. The abstract machine behind C was a revelation. I also read lots of UNIX and Plan 9 code: Ken Thompson's and Rob Pike's code certainly influenced STL. In any case, in 1987 C++ was not ready for STL and I had to move on.

At that time I discovered the works of Euler and my perception of the nature of mathematics underwent a dramatic transformation. I was de-Bourbakiized, stopped believing in sets, and was expelled from the Cantorian paradise. I still believe in abstraction, but now I know that one ends with abstraction, not starts with it. I learned that one has to adapt abstractions to reality and not the other way around. Mathematics stopped being a science of theories but reappeared to me as a science of numbers and shapes.

In 1993, after five years working on unrelated projects, I returned to generic programming. Andy Koenig suggested that I write a proposal for including my library into the C++ standard, Bjarne Stroustrup enthusiastically endorsed the proposal and in less than a year STL was accepted into the standard. STL is the result of 20 years of thinking but of less than two years of funding.

STL is only a limited success. While it became a widely used library, its central intuition did not get across. People confuse generic programming with using (and abusing) C++ templates. Generic programming is about abstracting and classifying algorithms and data structures. It gets its inspiration from Knuth and not from type theory. Its goal is the incremental construction of systematic catalogs of useful, efficient and abstract algorithms and data structures. Such an undertaking is still a dream.

You can find references to the work leading to STL at www.stepanovpapers.com.

I am more optimistic about the long-term impact of Alex's ideas than he is. However, we agree that the STL is just the first step of a long journey.

4.1.9 The impact of the STL

The impact of the STL on the thinking on C++ has been immense. Before the STL, I consistently listed three fundamental programming styles ("paradigms") as being supported by C++ [113]:

- Procedural programming
- Data abstraction
- Object-oriented programming

I saw templates as support for data abstraction. After playing with the STL for a while, I factored out a fourth style:

- Generic programming

The techniques based on the use of templates and largely inspired by techniques from functional programming are qualitatively different from traditional data abstraction. People simply think differently about types, objects, and resources. New C++ libraries are written — using templates — to be statically type safe and efficient. Templates are the key to embedded systems programming and high-performance numeric programming where resource management and correctness are key [67]. The STL itself is not always ideal in those areas. For example, it doesn't provide direct support for linear algebra and it can be tricky to use in hard-real-time systems where free store use is banned. However, the STL demonstrates what can be done with templates and gives examples of effective techniques. For example, the use of iterators (and allocators) to separate logical memory access from actual memory access is key to many high-performance numeric techniques [86, 96] and the use of small, easily inlined, objects is key to examples of optimal use of hardware in embedded systems programming. Some of these techniques are documented in the standards committee's technical report on performance (§6.1). The emphasis on the STL and on generic programming in the C++ community in the late 1990s and early 2000s is to a large extent a reaction — and a constructive alternative — to a trend in the larger software-development community towards overuse of "object-oriented" techniques relying excessively on class hierarchies and virtual functions.

Obviously, the STL isn't perfect. There is no one "thing" to be perfect relative to. However, it broke new ground and has had impact even beyond the huge C++ community (§9.3). It also inspired many to use templates both in more disciplined and more adventurous ways. People talk about "template meta-programming" (§7.2.2) and generative programming [31] and try to push the techniques pioneered by the STL beyond the STL. Another line of attack is to consider how C++ could better support effective uses of templates (concepts, auto, etc.; see §8).

Inevitably, the success of STL brought its own problems. People wanted to write all kinds of code in the STL style. However, like any other style or technique, the STL style or even generic programming in general isn't ideal for every kind of problem. For example, generic programming relying on templates and overloading completely resolves all name bindings at compile time. It does not provide a mechanism for bindings that are resolved at run time; that's what class hierarchies and their associated object-oriented design techniques are for. Like all successful lan-

guage mechanisms and programming techniques, templates and generic programming became fashionable and severely overused. Programmers built truly baroque and brittle constructs based on the fact that template instantiation and deduction is Turing complete. As I earlier observed for C++’s object-oriented facilities and techniques: “Just because you can do it, doesn’t mean that you have to”. Developing a comprehensive and simple framework for using the different programming styles supported by C++ is a major challenge for the next few years. As a style of programming, “multi-paradigm programming” [121] is underdeveloped. It often provides solutions that are more elegant and outperforms alternatives [28], but we don’t (yet) have a simple and systematic way of combining the programming styles. Even its name gives away its fundamental weakness.

Another problem with the STL is that its containers are non-intrusive. From the point of view of code clarity and independence of concepts, being non-intrusive is a huge advantage. However, it does mean that we need to copy elements into containers or insert objects with default values into containers and later give them the desired values. Sometimes that’s inefficient or inconvenient. For example, people tend not to insert large objects into a vector for that reason; instead, they insert pointers to such large objects. Similarly, the implicit memory management that the standard containers provide for their elements is a major convenience, but there are applications (e.g., in some embedded and high-performance systems) where such implicit memory management must be avoided. The standard containers provide features for ensuring that (e.g., `reserve`), but they have to be understood and used to avoid problems.

4.2 Other Parts of the Standard Library

From 1994 onwards, the STL dominated the work on the standard library and provided its major area of innovation. However, it was not the only area of work. In fact, the standard library provides several components:

- basic language run-time support (memory management, run-time type information (RTTI), exceptions, etc.)
- the C standard library
- the STL (containers, algorithms, iterators, function objects)
- iostreams (templatized on character type and implicitly on locale)
- locales (objects characterizing cultural preferences in I/O)
- string (templatized on character type)
- bitset (a set of bits with logical operations)
- complex (templatized on scalar type)
- valarray (templatized on scalar type)

- auto_ptr (a resource handle for objects templatized on type)

For a variety of reasons, the stories of the other library components are less interesting and less edifying than the story of the STL. Most of the time, work on each of these components progressed in isolation from work on the others. There was no overall design or design philosophy. For example, `bitset` is range checked whereas `string` isn’t. Furthermore, the design of several components (such as `string`, `complex`, and `iostream`) was constrained by compatibility concerns. Several (notably `iostream` and `locale`) suffered from the “second-system effect” as their designers tried to cope with all kinds of demands, constraints, and existing practice. Basically, the committee failed to contain “design by committee” so whereas the STL reflects a clear philosophy and coherent style, most of the other components suffered. Each represents its own style and philosophy, and some (such as `string`) manage simultaneously to present several. I think `complex` is the exception here. It is basically my original design [91] templatized to allow for a variety of scalar types:

```
complex<double> z;      // double-precision
complex<float> x;       // single-precision
complex<short> point;   // integer grid
```

It is hard to seriously mess up math.

The committee did have some serious discussions about the scope of the standard library. The context for this discussion was the small and universally accepted C standard library (which the C++ standard adopted with only the tiniest of modification) and the huge corporate foundation libraries. During the early years of the standards process, I articulated a set of guidelines for the scope of the C++ standard library:

First of all, the key libraries now in almost universal use must be standardized. This means that the exact interface between C++ and the C standard libraries must be specified and the `iostreams` library must be specified. In addition, the basic language support must be specified. . . .

Next, the committee must see if it can respond to the common demand for “more useful and standard classes,” such as `string`, without getting into a mess of design by committee and without competing with the C++ library industry. Any libraries beyond the C libraries and `iostreams` accepted by the committee must be in the nature of building blocks rather than more ambitious frameworks. The key role of a standard library is to ease communication between separately developed, more ambitious libraries.

The last sentence delineated the scope of the committee’s efforts. The elaboration of the requirement for the standard library to consist of building blocks for more ambitious libraries and frameworks emphasized absolute efficiency and

extreme generality. One example I frequently used to illustrate the seriousness of those demands was that a container where element access involved a virtual function call could not be sufficiently efficient and that a container that couldn't hold an arbitrary type could not be sufficiently general (see also §4.1.1). The committee felt that the role of the standard library was to support rather than supplant other libraries.

Towards the end of the work on the 1998 standard, there was a general feeling in the committee that we hadn't done enough about libraries, and also that there had been insufficient experimentation and too little attention to performance measurement for the libraries we did approve. The question was how to address those issues in the future. One — classical standards process — approach was to work on technical reports (see §6.2). Another was initiated by Beman Dawes in 1998, called Boost [16]. To quote from boost.org (August 2006):

"Boost provides free peer-reviewed portable C++ source libraries. We emphasize libraries that work well with the C++ Standard Library. Boost libraries are intended to be widely useful, and usable across a broad spectrum of applications. The Boost license encourages both commercial and non-commercial use. We aim to establish "existing practice" and provide reference implementations so that Boost libraries are suitable for eventual standardization. Ten Boost libraries are already included in the C++ Standards Committee's Library Technical Report (TR1) as a step toward becoming part of a future C++ Standard. More Boost libraries are proposed for the upcoming TR2".

Boost thrived and became a significant source of libraries and ideas for the standards committee and the C++ community in general.

5. Language Features: 1991-1998

By 1991, the most significant C++ language features for C++98 had been accepted: templates and exceptions as specified in the ARM were officially part of the language. However, work on their detailed specification went on for another several years. In addition, the committee worked on many new features, such as

- 1992** Covariant return types — the first extension beyond the features described in the ARM
- 1993** Run-time type identification (RTTI: `dynamic_cast`, `typeid`, and `type_info`); §5.1.2
- 1993** Declarations in conditions; §5.1.3
- 1993** Overloading based on enumerations
- 1993** Namespaces; §5.1.1
- 1993** `mutable`
- 1993** New casts (`static_cast`, `reinterpret_cast`, and `const_cast`)

1993 A Boolean type (`bool`); §5.1.4

1993 Explicit template instantiation

1993 Explicit template argument specification in function template calls

1994 Member templates ("nested templates")

1994 Class templates as template arguments

1996 In-class member initializers

1996 Separate compilation of templates (`export`); §5.2

1996 Template partial specialization

1996 Partial ordering of overloaded function templates

I won't go into detail here; the history of these features can be found in D&E [121] and TC++PL3 [126] describes their use. Obviously, most of these features were proposed and discussed long before they were voted into the standard.

Did the committee have overall criteria for acceptance of new features? Not really. The introduction of classes, class hierarchies, templates, and exceptions each (and in combination) represented a deliberate attempt to change the way people think about programming and write code. Such a major change was part of my aims for C++. However, as far as a committee can be said to think, that doesn't seem to be the way it does it. Individuals bring forward proposals, and the ones that make progress through the committee and reach a vote tend to be of limited scope. The committee members are busy and primarily practical people with little patience for abstract goals and a liking of concrete details that are amenable to exhaustive examination.

It is my opinion that the sum of the facilities added gives a more complete and effective support of the programming styles supported by C++, so we could say that the overall aim of these proposals is to "provide better support for procedural, object-oriented, and generic programming and for data abstraction". That's true, but it is not a concrete criterion that can be used to select proposals to work on from a long list. To the extent that the process has been successful in selecting new "minor features", it has been the result of decisions by individuals on a proposal-by-proposal basis. That's not my ideal, but the result could have been much worse. ISO C++ (C++98) is a better approximation to my ideals than the previous versions of C++ were. C++98 is a far more flexible (powerful) programming language than "ARM C++" (§3). The main reason for that is the cumulative effect of the refinements, such as member templates.

Not every feature accepted is in my opinion an improvement, though. For example, "in-class initialization of static const members of integral type with a constant expression" (proposed by John "Max" Skaller representing Australia and New Zealand) and the rule that `void f(T)` and `void f(const T)` denote the same function (proposed by Tom Plum for C compatibility reasons) share the dubious distinction of having been voted into C++ "over my dead body".

5.1 Some “Minor Features”

The “minor features” didn’t feel minor when the committee worked on them, and may very well not look minor to a programmer using them. For example, I refer to namespaces and RTTI as “major” in D&E [121]. However, they don’t significantly change the way we think about programs, so I will only briefly discuss a few of the features that many would deem “not minor”.

5.1.1 Namespaces

C provides a single global namespace for all names that don’t conveniently fit into a single function, a single struct, or a single translation unit. This causes problems with name clashes. I first grappled with this problem in the original design of C++ by defaulting all names to be local to a translation unit and requiring an explicit extern declaration to make them visible to other translation units. This idea was neither sufficient to solve the problem nor sufficiently compatible to be acceptable, so it failed.

When I devised the type-safe linkage mechanism [121], I reconsidered the problem. I observed that a slight change to the

```
extern "C" { /* ... */ }
```

syntax, semantics, and implementation technique would allow us to have

```
extern XXX { /* ... */ }
```

mean that names declared in XXX were in a separate scope XXX and accessible from other scopes only when qualified by XXX:: in exactly the same way static class members are accessed from outside their class.

For various reasons, mostly related to lack of time, this idea lay dormant until it resurfaced in the ANSI/ISO committee discussions early in 1991. First, Keith Rowe from Microsoft presented a proposal that suggested the notation

```
bundle XXX { /* ... */ };
```

as a mechanism for defining a named scope and an operator use for bringing all names from a bundle into another scope. This led to a — not very vigorous — discussion among a few members of the extensions group including Steve Dovich, Dag Brück, Martin O’Riordan, and me. Eventually, Volker Bauche, Roland Hartinger, and Erwin Unruh from Siemens refined the ideas discussed into a proposal that didn’t use new keywords:

```
:: XXX :: { /* ... */ };
```

This led to a serious discussion in the extensions group. In particular, Martin O’Riordan demonstrated that this :: notation led to ambiguities with :: used for class members and for global names.

By early 1993, I had — with the help of multi-megabyte email exchanges and discussions at the standards meetings

— synthesized a coherent proposal. I recall technical contributions on namespaces from Dag Brück, John Bruns, Steve Dovich, Bill Gibbons, Philippe Gautron, Tony Hansen, Peter Juhl, Andrew Koenig, Eric Krohn, Doug McIlroy, Richard Minner, Martin O’Riordan, John “Max” Skaller, Jerry Schwarz, Mark Terribile, Mike Vilot, and me. In addition, Mike Vilot argued for immediate development of the ideas into a definite proposal so that the facilities would be available for addressing the inevitable naming problems in the ISO C++ library. In addition to various common C and C++ techniques for limiting the damage of name clashes, the facilities offered by Modula-2 and Ada were discussed. Namespaces were voted into C++ at the Munich meeting in July 1993. So, we can write:

```
namespace XXX {  
    // ...  
    int f(int);  
}  
  
int f(int);  
int x = f(1);           // call global f  
int y = XXX::f(1);     // call XXX's f
```

At the San Jose meeting in November 1993, it was decided to use namespaces to control names in the standard C and C++ libraries.

The original namespace design included a few more facilities, such as namespace aliases to allow abbreviations for long names, using declarations to bring individual names into a namespace, using directives to make all names from a namespace available with a single directive. Three years later, argument-dependent lookup (ADL or “Koenig lookup”) was added to make namespaces of argument type names implicit.

The result was a facility that is useful and used but rarely loved. Namespaces do what they are supposed to do, sometimes elegantly, sometimes clumsily, and sometimes they do more than some people would prefer (especially argument-dependent lookup during template instantiation). The fact that the C++ standard library uses only a single namespace for all of its major facilities is an indication of a failure to establish namespaces as a primary tool of C++ programmers. Using sub-namespaces for the standard library would have implied a standardization of parts of the library implementation (to say which facilities were in which namespaces and which parts depended on other parts). Some library vendors strongly objected to such constraints on their traditional freedom as implementers — traditionally the internal organization of C and C++ libraries have been essentially unconstrained. Using sub-namespaces would also have been a source of verbosity. Argument-dependent lookup would have helped, but it was only introduced later in the standardization process. Also, ADL suffers from a bad interaction with templates that in some cases make it prefer a surpris-

ing template over an obvious non-template. Here "surprising" and "obvious" are polite renderings of user comments.

This has led to proposals for C++0x to strengthen namespaces, to restrict their use, and most interestingly a proposal from David Vandevoorde from EDG to make some namespaces into modules [146] — that is, to provide separately compiled namespaces that load as modules. Obviously, that facility looks a bit like the equivalent features of Java and C#.

5.1.2 Run-time type information

When designing C++, I had left out facilities for determining the type of an object (Simula's QUA and INSPECT similar to Smalltalk's isKindOf and isA). The reason was that I had observed frequent and serious misuse to the great detriment of program organization: people were using these facilities to implement (slow and ugly) versions of a switch statement.

The original impetus for adding facilities for determining the type of an object at run time to C++ came from Dmitry Lenkov from Hewlett-Packard. Dmitry in turn built on experience from major C++ libraries such as Interviews [81], the NIH library [50], and ET++ [152]. The RTTI mechanisms provided by these libraries (and others) were mutually incompatible, so they became a barrier to the use of more than one library. Also, all require considerable foresight from base class designers. Consequently, a language-supported mechanism was needed.

I got involved in the detailed design for such mechanisms as the coauthor with Dmitry of the original proposal to the committee and as the main person responsible for the refinement of the proposal in the committee [119]. The proposal was first presented to the committee at the London meeting in July 1991 and accepted at the Portland, Oregon meeting in March 1993.

The run-time type information mechanism consists of three parts:

- An operator, `dynamic_cast`, for obtaining a pointer to an object of a derived class given a pointer to a base class of that object. The operator `dynamic_cast` delivers that pointer only if the object pointed to really is of the specified derived class; otherwise it returns 0.
- An operator, `typeid`, for identifying the exact type of an object given a pointer to a base class.
- A structure, `type_info`, acting as a hook for further run-time information associated with a type.

Assume that a library supplies class `dialog_box` and that its interfaces are expressed in terms of `dialog_box`s. I, however, use both `dialog_box`s and my own `Sbox`s:

```
class dialog_box : public window {
    // library class known to ``the system''
public:
    virtual int ask();
    // ...
}
```

```
};

class Sbox : public dialog_box {
    // can be used to communicate a string
public:
    int ask();
    virtual char* get_string();
    // ...
};
```

So, when the system/library hands me a pointer to a `dialog_box`, how can I know whether it is one of my `Sbox`s? Note that I can't modify the library to know my `Sbox` class. Even if I could, I wouldn't, because then I would have to modify every new version of the library forever after. So, when the system passes an object to my code, I sometimes need to ask it if was "one of mine". This question can be asked directly using the `dynamic_cast` operator:

```
void my_fct(dialog_box* bp)
{
    if (Sbox* sbp = dynamic_cast<Sbox*>(bp)) {
        // use sbp
    }
    else {
        // treat *pb as a ``plain'' dialog box
    }
}
```

The `dynamic_cast<T*>(p)` converts `p` to the desired type `T*` if `*p` really is a `T` or a class derived from `T`; otherwise, the value of `dynamic_cast<T*>(p)` is 0. This use of `dynamic_cast` is the essential operation of a GUI callback. Thus, C++'s RTTI can be seen as the minimal facility for supporting a GUI.

If you don't want to test explicitly, you can use references instead of pointers:

```
void my_fct(dialog_box& br)
{
    Sbox& sbr = dynamic_cast<Sbox&>(br);
    // use sbr
}
```

Now, if the `dialog_box` isn't of the expected type, an exception is thrown. Error handling can then be elsewhere (§5.3).

Obviously, this run-time type information is minimal. This has led to requests for the maximal facility: a full meta-data facility (reflection). So far, this has been deemed unsuitable for a programming language that among other things is supposed to leave its applications with a minimal memory footprint.

5.1.3 Declarations in conditions

Note the way the cast, the declaration, and the test were combined in the "box example":

```
if (Sbox* sbp = dynamic_cast<Sbox*>(bp)) {
    // use sbp
}
```

This makes a neat logical entity that minimizes the chance of forgetting to test, minimizes the chance of forgetting to initialize, and limits the scope of the variable to its minimum. For example, the scope of `dbp` is the `if`-statement.

The facility is called “declaration in condition” and mirrors the “declaration as `for`-statement initializer”, and “declaration as statement”. The whole idea of allowing declarations everywhere was inspired by the elegant statements-as-expressions definition of Algol68 [154]. I was therefore most amazed when Charles Lindsey explained to me at the HOPL-II conference that Algol68 for technical reasons had not allowed declarations in conditions.

5.1.4 Booleans

Some extensions really are minor, but the discussions about them in the C++ community are not. Consider one of the most common enumerations:

```
enum bool { false, true };
```

Every major program has that one or one of its cousins:

```
#define bool char
#define Bool int
typedef unsigned int BOOL;
typedef enum { F, T } Boolean;
const true = 1;
#define TRUE 1
#define False (!True)
```

The variations are apparently endless. Worse, most variations imply slight variations in semantics, and most clash with other variations when used together.

Naturally, this problem has been well known for years. Dag Brück (representing Ericsson and Sweden) and Andrew Koenig (AT&T) decided to do something about it: “The idea of a Boolean data type in C++ is a religious issue. Some people, particularly those coming from Pascal or Algol, consider it absurd that C should lack such a type, let alone C++. Others, particularly those coming from C, consider it absurd that anyone would bother to add such a type to C++”

Naturally, the first idea was to define an `enum`. However, Dag Brück and Sean Corfield (UK) examined hundreds of thousands of lines of C++ and found that most Boolean types were used in ways that required implicit conversion of `bool` to and from `int`. C++ does not provide implicit conversion of `ints` to enumerations, so defining a standard `bool` as an enumeration would break too much existing code. So why bother with a Boolean type?

- The Boolean data type is a fact of life whether it is a part of a C++ standard or not.
- The many clashing definitions make it hard to use *any* Boolean type conveniently and safely.
- Many people want to overload based on a Boolean type.

Somewhat to my surprise, the committee accepted this argument, so `bool` is now a distinct integral type in C++ with

literals `true` and `false`. Non-zero values can be implicitly converted to `true`, and `true` can be implicitly converted to 1. Zero can be implicitly converted to `false`, and `false` can be implicitly converted to 0. This ensures a high degree of compatibility.

Over the years, `bool` proved popular. Unexpectedly, I found it useful in teaching C++ to people without previous programming experience. After `bool`’s success in C++, the C standards committee decided to also add it to C. Unfortunately, they decided to do so in a different and incompatible way, so in C99 [64], `bool` is a macro for the keyword `_Bool` defined in the header `<stdbool.h>` together with macros `true` and `false`.

5.2 The Export Controversy

From the earliest designs, templates were intended to allow a template to be used after specifying just a declaration in a translation unit [117, 35]. For example:

```
template<class In, class T>
In find(In, In, const T&); // no function body

vector<int>::iterator p =
    find(vi.begin(), vi.end(), 42);
```

It is then the job of the compiler and linker to find and use the definition of the `find` template (D&E §15.10.4). That’s the way it is for other language constructs, such as functions, but for templates that’s easily said but extremely hard to do.

The first implementation of templates, Cfront 3.0 (October 1991), implemented this, but in a way that was very expensive in both compile time and link time. However, when Taumetric and Borland implemented templates, they introduced the “include everything” model: Just place all template definitions in header files and the compiler plus linker will eliminate the multiple definitions you get when you include a file multiple times in separately compiled translation units. The First Borland compiler with “rudimentary template support” shipped November 20, 1991, quickly followed by version 3.1 and the much more robust version 4.0 in November 1993 [27]. Microsoft, Sun, and others followed along with (mutually incompatible) variations of the “include everything” approach. Obviously, this approach violates the usual separation between an implementation (using definitions) and an interface (presenting only declarations) and makes definitions vulnerable to macros, unintentional overload resolution, etc. Consider a slightly contrived example:

```
// printer.h:
template<class Destination>
class Printer {
    locale loc;
    Destination des;
public:
    template<class T> void out(const T& x)
        { print(des,x,loc); }
```

```
// ...
};
```

We might use Printer in two translation units like this:

```
//user1.c:
typedef int locale; // represent locale by int
#define print(a,b,c) a(c)<<x
#include "printer.h"
// ...
```

and this

```
//user2.c:
#include<locale> // use standard locale
using namespace std;
#include "printer.h"
// ...
```

This is obviously illegal because differences in the contexts in which printer.h are seen in user1.c and user2.c lead to inconsistent definitions of Printer. However, such errors are hard for a compiler to detect. Unfortunately, the probability of this kind of error is far higher in C++ than in C and even higher in C++ using templates than in C++ that doesn't use templates. The reason is that when we use templates there is so much more text in header files for typedefs, overloads, and macros to interfere with. This leads to defensive programming practices (such as naming all local names in template definitions in a cryptic style unlikely to clash, e.g., _L2) and a desire to reduce the amount of code in the header files through separate compilation.

In 1996, a vigorous debate erupted in the committee over whether we should not just accept the “include everything” model for template definitions, but actually outlaw the original model of separation of template declarations and definitions into separate translation units. The arguments of the two sides were basically

- Separate translation of templates is too hard (if not impossible) and such a burden should not be imposed on implementers
- Separate translation of templates is necessary for proper code organization (according to data-hiding principles)

Many subsidiary arguments supported both sides. Mike Ball (Sun) and John Spicer (EDG) led the “ban separate compilation of templates” group and Dag Bruck (Ericsson and Sweden) usually spoke for the “preserve separate compilation of templates” group. I was on the side that insisted on separate compilation of templates. As ever in really nasty discussions, both sides were mostly correct on their key points. In the end, people from SGI — notably John Wilkinson — proposed a new model that was accepted as a compromise. The compromise was named after the keyword used to indicate that a template could be separately translated: `export`.

The separate compilation of templates issue festers to this day: The “`export`” feature remains disabled even in some compilers that do support it because enabling it would break

ABIs. As late as 2003, Herb Sutter (representing Microsoft) and Tom Plum (of Plum Hall) proposed a change to the standard so that an implementation that didn't implement separate compilation of templates would still be conforming; that is, `export` would be an optional language feature. The reason given was again implementation complexity plus the fact that even five years after the standard was ratified only one implementation existed. That motion was defeated by an 80% majority, partly because an implementation of `export` now existed. Independently of the technical arguments, many committee members considered it unfair to deem a feature optional after some, but not all, implementers had spent significant time and effort implementing it.

The real heroes of this sad tale are the implementers of the EDG compiler: Steve Adamczyk, John Spicer, and David Vandevoorde. They strongly opposed separate compilation of templates, finally voted for the standard as the best compromise attainable, and then proceeded to spend more than a year implementing what they had opposed. That's professionalism! The implementation was every bit as difficult as its opponents had predicted, but it worked and provided some (though not all) of the benefits that its proponents had promised. Unfortunately, some of the restrictions on separately compiled templates that proved essential for the compromise ended up not providing their expected benefits and complicated the implementation. As ever, political compromises on technical issues led to “warts”.

I suspect that one major component of a better solution to the separate compilation of templates is concepts (§8.3.3) and another is David Vandevoorde's modules [146].

5.3 Exception Safety

During the effort to specify the STL we encountered a curious phenomenon: We didn't quite know how to talk about the interaction between templates and exceptions. Quite a few people were placing blame for this problem on templates and others began to consider exceptions fundamentally flawed (e.g., [20]) or at least fundamentally flawed in the absence of automatic garbage collection. However, when a group of “library people” (notably Nathan Myers, Greg Colvin, and Dave Abrahams) looked into this problem, they found that we basically had a language feature — exceptions — that we didn't know how to use well. The problem was in the interaction between resources and exceptions. If throwing an exception renders resources inaccessible there is no hope of recovering gracefully. I had of course considered this when I designed the exception-handling mechanisms and come up with the rules for exceptions thrown from constructors (correctly handling partially constructed composite objects) and the “resource acquisition is initialization” technique (§5.3.1). However, that was only a good start and an essential foundation. What we needed was a conceptual framework — a more systematic way of thinking about resource management. Together with many other people, notably Matt Austern, such a framework was developed.

Dave Abrahams condensed the result of work over a couple of years into three guarantees [1]:

- The *basic guarantee*: that the invariants of the component are preserved, and no resources are leaked.
- The *strong guarantee*: that the operation has either completed successfully or thrown an exception, leaving the program state exactly as it was before the operation started.
- The *no-throw guarantee*: that the operation will not throw an exception.

Note that the strong guarantee basically is the database “commit or rollback” rule. Using these fundamental concepts, the library working group described the standard library and implementers produced efficient and robust implementations. The standard library provides the basic guarantee for all operations with the caveat that we may not exit a destructor by throwing an exception. In addition, the library provides the strong guarantee and the no-throw guarantee for key operations. I found this result important enough to add an appendix to TC++PL [124], yielding [126]. For details of the standard library exception guarantees and programming techniques for using exceptions, see Appendix E of TC++PL.

The first implementations of the STL using these concepts to achieve exception safety were Matt Austern’s SGI STL and Boris Fomitch’s STLPort [42] augmented with Dave Abrahams’ exception-safe implementations of standard containers and algorithms. They appeared in the spring of 1997.

I think the key lesson here is that it is not sufficient just to know how a language feature behaves. To write good software, we must have a clearly articulated design strategy for problems that require the use of the feature.

5.3.1 Resource management

Exceptions are typically — and correctly — seen as a control structure: a `throw` transfers control to some `catch`-clause. However, sequencing of operations is only part of the picture: error handling using exceptions is mostly about resource management and invariants. This view is actually built into the C++ class and exception primitives in a way that provides a necessary foundation for the guarantees and the standard library design.

When an exception is thrown, every constructed object in the path from the `throw` to the `catch` is destroyed. The destructors for partially constructed objects (and unconstructed objects) are not invoked. Without those two rules, exception handling would be unmanageable (in the absence of other support). I (clumsily) named the basic technique “resource acquisition is initialization” — commonly abbreviated to “RAII”. The classical example [118] is

```
// naive and unsafe code:
void use_file(const char* fn)
```

```
{
    FILE* f = fopen(fn, "w"); // open file fn
    // use f
    fclose(f); // close file fn
}
```

This looks plausible. However, if something goes wrong after the call of `fopen` and before the call of `fclose`, an exception may cause `use_file` to be exited without calling `fclose`. Please note that exactly the same problem can occur in languages that do not support exception handling. For example, a call of the standard C library function `longjmp` would have the same bad effects. Even a misguided return among the code using `f` would cause the program to leak a file handle. If we want to support writing fault-tolerant systems, we must solve this problem.

The general solution is to represent a resource (here the file handle) as an object of some class. The class’ constructor acquires the resource and the class’ destructor gives it back. For example, we can define a class `File_ptr` that acts like a `FILE*`:

```
class File_ptr {
    FILE* p;
public:
    File_ptr(const char* n, const char* a)
    {
        p = fopen(n,a);
        if (p==0) throw Failed_to_open(n);
    }
    ~File_ptr() { fclose(p); }
    // copy, etc.
    operator FILE*() { return p; }
};
```

We can construct a `File_ptr` given the arguments required for `fopen`. The `File_ptr` will be destroyed at the end of its scope and its destructor closes the file. Our program now shrinks to this minimum

```
void use_file(const char* fn)
{
    File_ptr f(fn, "r"); // open file fn
    // use f
} // file fn implicitly closed
```

The destructor will be called independently of whether the function is exited normally or because an exception is thrown.

The general form of the problem looks like this:

```
void use()
{
    // acquire resource 1
    // ...
    // acquire resource n

    // use resources

    // release resource n
    // ...
```

```
// release resource 1
}
```

This applies to any “resource” where a resource is anything you acquire and have to release (hand back) for a system to work correctly. Examples include files, locks, memory, iostream states, and network connections. Please note that automatic garbage collection is *not* a substitute for this: the point in time of the release of a resource is often important logically and/or performance-wise.

This is a systematic approach to resource management with the important property that correct code is shorter and less complex than faulty and primitive approaches. The programmer does not have to remember to do anything to release a resource. This contrasts with the older and ever-popular `finally` approach where a programmer provides a `try-block` with code to release the resource. The C++ variant of that solution looks like this:

```
void use_file(const char* fn)
{
    FILE* f = fopen(fn, "r"); // open file fn
    try {
        // use f
    }
    catch (...) { // catch all
        fclose(f); // close file fn
        throw; // re-throw
    }
    fclose(f); // close file fn
}
```

The problem with this solution is that it is verbose, tedious, and potentially expensive. It can be made less verbose and tedious by providing a `finally` clause in languages such as Java, C#, and earlier languages [92]. However, shortening such code doesn’t address the fundamental problem that the programmer has to remember to write release code for each acquisition of a resource rather just once for each resource, as in the RAII approach. The introduction of exceptions into the ARM and their presentation as a conference paper [79] was delayed for about half a year until I found “resource acquisition is initialization” as a systematic and less error-prone alternative to the `finally` approach.

5.4 Automatic Garbage Collection

Sometime in 1995, it dawned on me that a majority of the committee was of the opinion that plugging a garbage collector into a C++ program was not standard-conforming because the collector would inevitably perform some action that violated a standard rule. Worse, they were obviously right about the broken rules. For example:

```
void f()
{
    int* p = new int[100];
    // fill *p with valuable data
    file << p; // write the pointer to a file
    p = 0; // remove the pointer to the ints
```

```
// work on something else for a week
file >> p;
if (p[37] == 5) { // now use the ints
    // ...
}
```

My opinion — as expressed orally and in print — was roughly: “such programs deserve to be broken” and “it is perfectly good C++ to use a conservative garbage collector”. However, that wasn’t what the draft standard said. A garbage collector would undoubtedly have recycled that memory before we read the pointer back from the file and started using the integer array again. However, in standard C and standard C++, there is absolutely nothing that allows a piece of memory to be recycled without some explicit programmer action.

To fix this problem, I made a proposal to explicitly allow “optional automatic garbage collection” [123]. This would bring C++ back to what I had thought I had defined it to be and make the garbage collectors already in actual use [11, 47] standard conforming. Explicitly mentioning this in the standard would also encourage use of GC where appropriate. Unfortunately, I seriously underestimated the dislike of garbage collection in a large section of the committee and also mishandled the proposal.

My fatal mistake with the GC proposal was to get thoroughly confused about the meaning of “optional”. Did “optional” mean that an implementation didn’t have to provide a garbage collector? Did it mean that the programmer could decide whether the garbage collector was turned on or not? Was the choice made at compile time or run time? What should happen if I required the garbage collector to be activated and the implementation didn’t supply one? Can I ask if the garbage collector is running? How? How can I make sure that the garbage collector isn’t running during a critical operation? By the time a confused discussion of such questions had broken out and different people had found conflicting answers attractive, the proposal was effectively dead.

Realistically, garbage collection wouldn’t have passed in 1995, even if I hadn’t gotten confused. Parts of the committee

- strongly distrusted GC for performance reasons
- disliked GC because it was seen as a C incompatibility
- didn’t feel they understood the implications of accepting GC (we didn’t)
- didn’t want to build a garbage collector
- didn’t want to pay for a garbage collector (in terms of money, space, or time)
- wanted alternative styles of GC
- didn’t want to spend precious committee time on GC

Basically, it was too late in the standards process to introduce something that major. To get anything involving garbage collection accepted, I should have started a year earlier.

My proposal for garbage collection reflected the then major use of garbage collection in C++ — that is, conservative collectors that don’t make assumptions about which memory locations contain pointers and never move objects around in memory [11]. Alternative approaches included creating a type-safe subset of C++ so that it is possible to know exactly where every pointer is, using smart pointers [34] and providing a separate operator (`gcnew` or `new(gc)`) for allocating objects on a “garbage-collected heap”. All three approaches are feasible, provide distinct benefits, and have proponents. This further complicates any effort to standardize garbage collection for C++.

A common question over the years has been: Why don’t you add GC to C++? Often, the implication (or follow-up comment) is that the C++ committee must be a bunch of ignorant dinosaurs not to have done so already. First, I observe that in my considered opinion, C++ would have been still-born had it relied on garbage collection when it was first designed. The overheads of garbage collection at the time, on the hardware available, precluded the use of garbage collection in the hardware-near and performance-critical areas that were C++’s bread and butter. There were garbage-collected languages then, such as Lisp and Smalltalk, and people were reasonably happy with those for the applications for which they were suitable. It was not my aim to replace those languages in their established application areas. The aim of C++ was to make object-oriented and data-abstraction techniques affordable in areas where these techniques at the time were “known” to be impractical. The core areas of C++ usage involved tasks, such as device drivers, high-performance computation, and hard-real-time tasks, where garbage collection was (and is) either infeasible or not of much use.

Once C++ was established without garbage collection and with a set of language features that made garbage collection difficult (pointers, casts, unions, etc.), it was hard to retrofit it without doing major damage. Also, C++ provides features that make garbage collection unnecessary in many areas (scoped objects, destructors, facilities for defining containers and smart pointers, etc.). That makes the case for garbage collection less compelling.

So, why would I like to see garbage collection supported in C++? The practical reason is that many people write software that uses the free store in an undisciplined manner. In a program with hundreds of thousands of lines of code with news and deletes all over the place, I see no hope for avoiding memory leaks and access through invalid pointers. My main advice to people who are starting a project is simply: “don’t do that!”. It is fairly easy to write correct and efficient C++ code that avoids those problems through the use of containers (STL or others; §4.1), resource handles (§5.3.1, and (if needed) smart pointers (§6.2). However, many of us have to deal with older code that does deal with memory in an undisciplined way, and for such code plugging in a conservative garbage collector is often the best option. I expect

C++0x to require every C++ implementation to be shipped with a garbage collector that, somehow, can be either active or not.

The other reason that I suspect a garbage collector will eventually become necessary is that I don’t see how to achieve perfect type safety without one — at least not without pervasive testing of pointer validity or damaging compatibility (e.g. by using two-word non-local pointers). And improving type safety (e.g., “eliminate every implicit type violation” [121]) has always been a fundamental long-term aim of C++. Obviously, this is a very different argument from the usual “when you have a garbage collector, programming is easy because you don’t have to think about deallocation”. To contrast, my view can be summarized as “C++ is such a good garbage-collected language because it creates so little garbage that needs to be collected”. Much of the thinking about C++ has been focused on resources in general (such as locks, file handles, thread handles, and free store memory). As noted in §5.3, this focus has left traces in the language itself, in the standard library, and in programming techniques. For large systems, for embedded systems, and for safety-critical systems a systematic treatment of resources seems to me much more promising than a focus on garbage collection.

5.5 What Wasn’t Done

Choosing what to work on is probably more significant than how that work is done. If someone — in this case the C++ standards committee — decides to work on the wrong problem, the quality of the work done is largely irrelevant. Given the limited time and resources, the committee could cope with only a few topics and choose to work hard on those few rather than take on more topics. By and large, I think that the committee chose well and the proof of that is that C++98 is a significantly better language than ARM C++. Naturally, we could have done better still, but even in retrospect it is hard to know how. The decisions made at the time were taken with as much information (about problems, resources, possible solutions, commercial realities, etc.) as we had available then and who are we to second guess today?

Some questions are obvious, though:

- Why didn’t we add support for concurrency?
- Why didn’t we provide a much more useful library?
- Why didn’t we provide a GUI?

All three questions were seriously considered and in the first two cases settled by explicit vote. The votes were close to unanimous. Given that we had decided not to pursue concurrency or to provide a significantly larger library, the question about a GUI library was moot.

Many of us — probably most of the committee members — would have liked some sort of concurrency support. Concurrency is fundamental and important. Among other suggestions, we even had a pretty solid proposal for concur-

rency support in the form of micro-C++ from the University of Toronto [18]. Some of us, notably Dag Brück relying on data from Ericsson, looked at the issue and presented the case for *not* dealing with concurrency in the committee:

- We found the set of alternative ways of supporting concurrency large and bewildering.
- Different application areas apparently had different needs and definitely had different traditions.
- The experience with Ada's direct language support for concurrency was discouraging.
- Much (though of course not all) could be done with libraries.
- The committee lacked sufficient experience to do a solid design.
- We didn't want a set of primitives that favored one approach to concurrency over others.
- We estimated that the work — if feasible at all — would take years given our scarce resources.
- We wouldn't be able to decide what other ideas for improvements to drop to make room for the concurrency work.

My estimate at the time was that “concurrency is as big a topic as all of the other extensions we are considering put together”. I do not recall hearing what I in retrospect think would have been “the killer argument”: Any sufficient concurrency support will involve the operating system; since C++ is a systems programming language, we need to be able to map the C++ concurrency facilities to the primitives offered. But for each platform the owners insist that C++ programmers can use every facility offered. In addition, as a fundamental part of a multi-language environment, C++ cannot rely on a concurrency model that is dramatically different from what other languages support. The resulting design problem is so constrained that it has no solution.

The reason the lack of concurrency support didn't hurt the C++ community more than it did is that much of what people actually do with concurrency is pretty mundane and can be done through library support and/or minor (non-standard) language extensions. The various threads libraries (§8.6) and MPI [94] [49] offer examples.

Today, the tradeoffs appear to be different: The continuing increase in gate counts paired with the lack of increase of hardware clock speeds is a strong incentive to exploit low-level concurrency. In addition, the growth of multiprocessors and clusters requires other styles of concurrency support and the growth of wide-area networking and the web makes yet other styles of concurrent systems essential. The challenge of supporting concurrency is more urgent than ever and the major players in the C++ world seem far more open to the need for changes to accommodate it. The work on C++0x reflects that (§8.2).

The answer to “Why didn't we provide a much more useful library?” is simpler: We didn't have the resources (time and people) to do significantly more than we did. Even the STL caused a year's delay and gaining consensus on other components, such as `iostreams`, strained the committee. The obvious shortcut — adopting a commercial foundation library — was considered. In 1992, Texas Instruments offered their very nice library for consideration and within an hour five representatives of major corporations made it perfectly clear that if this offer was seriously considered they would propose their own corporate foundation libraries. This wasn't a way that the committee could go. Another committee with a less consensus-based culture might have made progress by choosing one commercial library over half-a-dozen others in major use, but not this C++ committee.

It should also be remembered that in 1994, many already considered the C++ standard library monstrously large. Java had not yet changed programmers' perspective of what they could expect “for free” from a language. Instead, many in the C++ community used the tiny C standard library as the measure of size. Some national bodies (notably the Netherlands and France) repeatedly expressed worry that C++ standard library was seriously bloated. Like many in the committee, I also hoped that the standard would help rather than try to supplant the C++ libraries industry.

Given those general concerns about libraries, the answer to “Why didn't we provide a GUI?” is obvious: The committee couldn't do it. Even before tackling the GUI-specific design issues, the committee would have had to tackle concurrency and settle on a container design. In addition, many of the members were simply blindsided. GUI was seen as just another large and complex library that people — given `dynamic_cast` (§5.1.2) — could write themselves (in particular, that was my view). They did. The problem today is not that there is no C++ GUI library, but that there are on the order of 25 such libraries in use (e.g., Gtkmm [161], FLTK [156], SmartWin++ [159], MFC [158], WTL [160], vxWidgets (formerly wxWindows) [162], Qt [10]). The committee could have worked on a GUI library, worked on library facilities that could be used as a basis for GUI libraries, or worked on standard library interfaces to common GUI functionality. The latter two approaches might have yielded important results, but those paths weren't taken and I don't think that the committee then had the talents necessary for success in that direction. Instead, the committee worked hard on more elaborate interfaces to stream I/O. That was probably a dead end because the facilities for multiple character sets and locale dependencies were not primarily useful in the context of traditional data streams.

6. Standards Maintenance: 1997-2005

After a standard is passed, the ISO process can go into a “maintenance mode” for up to five years. The C++ committee decided to do that because:

- the members were tired (after ten years' work for some members) and wanted to do something else,
- the community was way behind in understanding the new features,
- many implementers were way behind with the new features, libraries, and support tools,
- no great new ideas were creating a feeling of urgency among the members or in the community,
- for many members, resources (time and money) for standardization were low, and
- many members (including me) thought that the ISO process required a “cooling-off period”.

In “maintenance mode,” the committee primarily responded to defect reports. Most defects were resolved by clarifying the text or resolving contradictions. Only very rarely were new rules introduced and real innovation was avoided. Stability was the aim. In 2003, all these minor corrections were published under the name “Technical Corrigendum 1”. At the same time, members of the British national committee took the opportunity to remedy a long-standing problem: they convinced Wiley to publish a printed version of the (revised) standard [66]. The initiative and much of the hard work came from Francis Glassborow and Lois Goldthwaite with technical support from the committee’s project editor, Andrew Koenig, who produced the actual text.

Until the publication of the revised standard in 2003, the only copies of the standard available to the public were a very expensive (about \$200) paper copy from ISO or a cheap (\$18) pdf version from INCITS (formerly ANSI X3). The pdf version was a complete novelty at the time. Standards bodies are partially financed through the sales of standards, so they are most reluctant to make them available free of charge or cheaply. In addition, they don’t have retail sales channels, so you can’t find a national or international standard in your local book store — except the C++ standard, of course. Following the C++ initiative, the C standard is now also available.

Much of the best standards work is invisible to the average programmer and appears quite esoteric and often boring when presented. The reason is that a lot of effort is expended in finding ways of expressing clearly and completely “what everyone already knows, but just happens not to be spelled out in the manual” and in resolving obscure issues that — at least in theory — don’t affect most programmers. The maintenance is mostly such “boring and esoteric” issues. Furthermore, the committee necessarily focuses on issues where the standard contradicts itself — or appears to do so. However, these issues are essential to implementers trying to ensure that a given language use is correctly handled. In turn, these issues become essential to programmers because even the most carefully written large program will deliberately or accidentally depend on some feature that would appear ob-

scure or esoteric to some. Unless implementers agree, the programmer has a hard time achieving portability and easily becomes the hostage of a single compiler purveyor — and that would be contrary to my view of what C++ is supposed to be.

To give an idea of the magnitude of this maintenance task (which carries on indefinitely): Since the 1998 standard until 2006, the core and library working groups has each handled on the order of 600 “defect reports”. Fortunately, not all were real defects, but even determining that there really is no problem, or that the problem is just lack of clarity in the standard’s text, takes time and care.

Maintenance wasn’t all that the committee did from 1997 to 2003. There was a modest amount of planning for the future (thinking about C++0x), but the main activities were writing a technical report on performance issues [67] and one on libraries [68].

6.1 The Performance TR

The performance technical report (“TR”) [67] was prompted by a suggestion to standardize a subset of C++ for embedded systems programming. The proposal, called Embedded C++ [40] or simply EC++, originated from a consortium of Japanese embedded systems tool developers (including Toshiba, Hitachi, Fujitsu, and NEC) and had two main concerns: removal of language features that potentially hurt performance and removal of language features perceived to be too complicated for programmers (and thus seen as potential productivity or correctness hazards). A less clearly stated aim was to define something that in the short term was easier to implement than full standard C++.

The features banned in this (almost) subset included: multiple inheritance, templates, exceptions, run-time type information (§5.1.2), new-style casts, and name spaces. From the standard library, the STL and locales were banned and an alternative version of iostreams provided. I considered the proposal misguided and backwards looking. In particular, the performance costs were largely imaginary, or worse. For example, the use of templates has repeatedly been shown to be key to both performance (time and space) and correctness of embedded systems. However, there wasn’t much hard data in this area in 1996 when EC++ was first proposed. Ironically, it appears that most of the few people who use EC++ today use it in the form of “Extended EC++” [36], which is EC++ plus templates. Similarly, namespaces (§5.1.1) and new style casts (§5) are features that are primarily there to clarify code and can be used to ease maintenance and verification of correctness. The best documented (and most frequently quoted) overhead of “full C++” as compared to EC++ was iostreams. The primary reason for that is that the C++98 iostreams support locales whereas that older iostreams do not. This is somewhat ironic because the locales were added to support languages different from English (most notably Japanese) and can be optimized away in environments where they are not used (see [67]).

After serious consideration and discussion, the ISO committee decided to stick to the long-standing tradition of not endorsing dialects — even dialects that are (almost) subsets. Every dialect leads to a split in the user community, and so does even a formally defined subset when its users start to develop a separate culture of techniques, libraries, and tools. Inevitably, myths about failings of the full language relative to the “subset” will start to emerge. Thus, I recommend against the use of EC++ in favor of using what is appropriate from (full) ISO Standard C++.

Obviously, the people who proposed EC++ were right in wanting an efficient, well-implemented, and relatively easy-to-use language. It was up to the committee to demonstrate that ISO Standard C++ was that language. In particular, it seemed a proper task for the committee to document the utility of the features rejected by EC++ in the context of performance-critical, resource-constrained, or safety-critical tasks. It was therefore decided to write a technical report on “performance”. Its executive summary reads:

‘The aim of this report is:

- to give the reader a model of time and space overheads implied by use of various C++ language and library features,
- to debunk widespread myths about performance problems,
- to present techniques for use of C++ in applications where performance matters, and
- to present techniques for implementing C++ Standard language and library facilities to yield efficient code.

As far as run-time and space performance is concerned, if you can afford to use C for an application, you can afford to use C++ in a style that uses C++’s facilities appropriately for that application.

Not every feature of C++ is efficient and predictable to the extent that we need for some high-performance and embedded applications. A feature is predictable if we can in advance easily and precisely determine the time needed for each use. In the context of an embedded system, we must consider if we can use

- free store (`new` and `delete`)
- run-time type identification (`dynamic_cast` and `typeid`)
- exceptions (`throw` and `catch`)

The time needed to perform one of these operations can depend on the context of the code (e.g. how much stack unwinding a `throw` must perform to reach its matching `catch`) or on the state of the program (e.g. the sequence of `new`s and `delete`s before a `new`).

Implementations aimed at embedded or high performance applications all have compiler options for disabling

run-time type identification and exceptions. Free store usage is easily avoided. All other C++ language features are predictable and can be implemented optimally (according to the zero-overhead principle; §2). Even exceptions can be (and tend to be) efficient compared to alternatives [93] and should be considered for all but the most stringent hard-real-time systems. The TR discusses these issues and also defines an interface to the lowest accessible levels of hardware (such as registers). The performance TR was written by a working group primarily consisting of people who cared about embedded systems, including members of the EC++ technical committee. I was active in the performance working group and drafted significant portions of the TR, but the chairman and editor was first Martin O’Riordan and later Lois Goldthwaite. The acknowledgments list 28 people, including Jan Kristofferson, Dietmar Kühl, Tom Plum, and Detlef Vollmann. In 2004, that TR was approved by unanimous vote.

In 2004, after the TR had been finalized, Mike Gibbs from Lockheed-Martin Aero found an algorithm that allows `dynamic_cast` to be implemented in constant time, and fast [48]. This offers hope that `dynamic_cast` will eventually be usable for hard-real-time programming.

The performance TR is one of the few places where the immense amount of C++ usage in embedded systems surfaces in publicly accessible writing. This usage ranges from high-end systems such as found in telecommunications systems to really low-level systems where complete and direct access to specific hardware features is essential (§7). To serve the latter, the performance TR contains a “hardware addressing interface” together with guidelines for its usage. This interface is primarily the work of Jan Kristofferson (representing Ramtex International and Denmark) and Detlef Vollmann (representing Vollmann Engineering GmbH and Switzerland). To give a flavor, here is code copying a register buffer specified by a port called `PortA2_T`:

```
unsigned char mybuf[10];
register_buffer<PortA2_T, Platform> p2;
for (int i = 0; i != 10; ++i)
{
    mybuf[i] = p2[i];
}
```

Essentially the same operation can be done as a block read:

```
register_access<PortA3_T, Platform> p3;
UCharBuf myBlock;
myBlock = p3;
```

Note the use of templates and the use of integers as template arguments; it’s essential for a library that needs to maintain optimal performance in time and space. This comes as a surprise to many who have been regaled with stories of memory bloat caused by templates. Templates are usually implemented by generating a copy of the code used for each specialization; that is, for each combination of template arguments. Thus, obviously, if your code generated by a template takes up a lot of memory, you can use a lot of memory.

However, many modern templates are written based on the observation that an inline function may shrink to something that's as small as the function preamble or smaller still. That way you can simultaneously save both time and space. In addition to good inlining, there are two more bases for good performance from template code: dead code elimination and avoidance of spurious pointers.

The standard requires that no code is generated for a member function of a class template unless that member function is called for a specific set of template arguments. This automatically eliminates what could have become “dead code”. For example:

```
template<class T> class X {
public:
    void f() { /* ... */ }
    void g() { /* ... */ }
    // ...
};

int main()
{
    X<int> xi;
    xi.f();
    X<double> xd;
    xd.g();
    return 0;
}
```

For this program, the compiler must generate code for `X<int>::f()` and `X<double>::g()` but may not generate code for `X<int>::g()` and `X<double>::f()`. This rule was added to the standard in 1993, at my insistence, specifically to reduce code bloat, as observed in early uses of templates. I have seen this used in embedded systems in the form of the rule “all classes must be templates, even if only a single instantiation is used”. That way, dead code is automatically eliminated.

The other simple rule to follow to get good memory performance from templates is: “don't use pointers if you don't need them”. This rule preserves complete type information and allows the optimizer to perform really well (especially when inline functions are used). This implies that function templates that take simple objects (such as function objects) as arguments should do so by value rather than by reference. Note that pointers to functions and virtual functions break this rule, causing problems for optimizers.

It follows that to get massive code bloat, say megabytes, what you need is to

1. use large function templates (so that the code generated is large)
2. use lots of pointers to objects, virtual functions, and pointers to functions (to neuter the optimizer)
3. use “feature rich” hierarchies (to generate a lot of potentially dead code)

4. use a poor compiler and a poor optimizer

I designed templates specifically to make it easy for the programmer to avoid (1) and (2). Based on experience, the standard deals with (3), except when you violate (1) or (2). In the early 1990s, (4) became a problem. Alex Stepanov named it “the abstraction penalty” problem. He defined “the abstraction penalty” as the ratio of runtime between a templated operation (say, `find` on a `vector<int>`) and the trivial non-templated equivalent (say a loop over an array of `int`). An implementation that does all of the easy and obvious optimizations gets a ratio of 1. Poor compilers had an abstraction penalty of 3, though even then good implementations did significantly better. In October 1995, to encourage implementers to do better, Alex wrote the “abstraction penalty benchmark”, which simply measured the abstraction penalty [102]. Compiler and optimizer writers didn't like their implementations to be obviously poor, so today ratios of 1.02 or so are common.

The other — and equally important — aspect of C++'s support for embedded systems programming is simply that its model of computation and memory is that of real-world hardware: the built-in types map directly to memory and registers, the built-in operations map directly to machine operations, and the composition mechanisms for data structures do not impose spurious indirections or memory overhead [131]. In this, C++ equals C. See also §2.

The views of C++ as a close-to-machine language with abstraction facilities that can be used to express predicable type-safe low-level facilities has been turned into a coding standard for safety-critical hard-real-time code by Lockheed-Martin Aero [157]. I helped draft that standard. Generally, C and assembly language programmers understand the direct mapping of language facilities to hardware, but often not the the need for abstraction mechanisms and strong type checking. Conversely, programmers brought up with higher-level “object-oriented” languages often fail to see the need for closeness to hardware and expect some unspecified technology to deliver their desired abstractions without unacceptable overheads.

6.2 The Library TR

When we finished the standard in 1997, we were fully aware that the set of standard libraries was simply the set that we had considered the most urgently needed and also ready to ship. Several much-wanted libraries, such as hash tables, regular expression matching, directory manipulation, and threads, were missing. Work on such libraries started immediately in the Libraries Working Group chaired by Matt Austern (originally working at SGI with Alex Stepanov, then at AT&T Labs with me, and currently at Google). In 2001, the committee started work on a technical report on libraries. In 2004 that TR [68] specifying libraries that people considered most urgently needed was approved by unanimous vote.

Despite the immense importance of the standard library and its extensions, I will only briefly list the new libraries here:

- Polymorphic function object wrapper
- Tuple types
- Mathematical special functions
- Type traits
- Regular expressions
- Enhanced member pointer adaptor
- General-purpose smart pointers
- Extensible random number facility
- Reference wrapper
- Uniform method for computing function-object return types
- Enhanced binder
- Hash tables

Prototypes or industrial-strength implementations of each of these existed at the time of the vote; they are expected to ship with every new C++ implementation from 2006 onwards. Many of these new library facilities are obviously “technical”; that is, they exist primarily to support library builders. In particular, they exist to support builders of standard library facilities in the tradition of the STL. Here, I will just emphasize three libraries that are of direct interest to large numbers of application builders:

- Regular expressions
- General-purpose smart pointers
- Hash tables

Regular expression matching is one of the backbones of scripting languages and of much text processing. Finally, C++ has a standard library for that. The central class is `regex`, providing regular expression matching of patterns compatible with ECMAScript (formerly JavaScript or Jscript) and with other popular notations.

The main “smart pointer” is a reference-counted pointer, `shared_ptr`, intended for code where shared ownership is needed. When the last `shared_ptr` to an object is destroyed, the object pointed to is deleted. Smart pointers are popular, but not universally so and concerns about their performance and likely overuse kept `smart_ptr`’s “ancestor”, `counted_ptr`, out of C++98. Smart pointers are not the panacea they are sometimes presented to be. In particular, they can be far more expensive to use than ordinary pointers, destructors for objects “owned” by a set of `shared_ptr`s will run at unpredictable times, and if a lot of objects are deleted at once because the last `shared_ptr` to them is deleted you can incur “garbage-collection delays” exactly as if you were running a general collector. The costs primarily relate to free-store allocation of use-count objects and espe-

cially to locking during access to the use counts in threaded systems (“lock-free” implementations appear to help here). If it is garbage collection you want, you might be better off simply using one of the available garbage collectors [11, 47] or waiting for C++0x (§5.4).

No such worries affected hash tables; they would have been in C++98 had we had the time to do a proper detailed design and specification job. There was no doubt that a `hash_map` was needed as an alternative to `map` for large tables where the key was a character string and we could design a good hash function. In 1995, Javier Barreiro, Robert Fraley and David Musser tried to get a proposal ready in time for the standard and their work became the basis for many of the later `hash_maps` [8]. The committee didn’t have the time, though, and consequently the Library TR’s `unordered_map` (and `unordered_set`) are the result of about eight years of experiment and industrial use. A new name, “`unordered_map`”, was chosen because now there are half a dozen incompatible `hash_map`s in use. The `unordered_map` is the result of a consensus among the `hash_map` implementers and their key users in the committee. An `unordered_map` is “unordered” in the sense that an iteration over its elements are not guaranteed to be in any particular order: a hash function doesn’t define an ordering in the way a `map`’s `<` does.

The most common reaction to these extensions among developers is “that was about time; why did it take you so long?” and “I want much more right now”. That’s understandable (I too want much more right now — I just know that I can’t get it), but such statements reflect a lack of understanding what an ISO committee is and can do. The committee is run by volunteers and requires both a consensus and an unusual degree of precision in our specifications (see D&E §6.2). The committee doesn’t have the millions of dollars that commercial vendors can and do spend on “free”, “standard” libraries for their customers.

7. C++ in Real-World Use

Discussions about programming languages typically focus on language features: which does the language have? how efficient are they? More enlightened discussions focus on more difficult questions: how is the language used? how can it be used? who can use it? For example, in an early OOPSLA keynote, Kristen Nygaard (of Simula and OOP fame) observed: “if we build languages that require a PhD from MIT to use, we have failed”. In industrial contexts, the first — and often only — questions are: Who uses the language? What for? Who supports it? What are the alternatives? This section presents C++ and its history from the perspective of its use.

Where is C++ used? Given the number of users (§1), it is obviously used in a huge number of places, but since most of the use is commercial it is difficult to document. This is one of the many areas where the lack of a central

organization for C++ hurts the C++ community — nobody systematically gathers information about its use and nobody has anywhere near complete information. To give an idea of the range of application areas, here are a few examples where C++ is used for crucial components of major systems and applications:

- Adobe — Acrobat, Photoshop, Illustrator, ...
- Amadeus — airline reservations
- Amazon — e-commerce
- Apple — iPod interface, applications, device drivers, finder, ...
- AT&T — 1-800 service, provisioning, recovery after network failure, ...
- eBay — online auctions
- Games — Doom3, StarCraft, Halo, ...
- Google — search engines, Google Earth, ...
- IBM — K42 (very high-end operating system), AS/400, ...
- Intel — chip design and manufacturing, ...
- KLA-Tencor — semiconductor manufacturing
- Lockheed-Martin Aero — airplane control (F16, JSF), ...
- Maeslant Barrier — Dutch surge barrier control
- MAN B&W — marine diesel engine monitoring and fuel injection control
- Maya — professional 3D animation
- Microsoft — Windows XP, Office, Internet explorer, Visual Studio, .Net, C# compiler, SQL, Money, ...
- Mozilla Firefox — browser
- NASA/JPL — Mars Rover scene analysis and autonomous driving, ...
- Southwest Airlines — customer web site, flight reservations, flight status, frequent flyer program, ...
- Sun — compilers, OpenOffice, HotSpot Java Virtual Machine, ...
- Symbian — OS for hand-held devices (especially cellular phones)
- Vodafone — mobile phone infrastructure (including billing and provisioning)

For more examples, see [137]. Some of the most widely used and most profitable software products ever are on this list. Whatever C++'s theoretical importance and impact, it certainly met its most critical design aim: it became an immensely useful practical tool. It brought object-oriented programming and more recently also generic programming into the mainstream. In terms of numbers of applications and the range of application areas, C++ is used beyond any

individual's range of expertise. That vindicates my emphasis on generality in D&E [121] (Chapter 4).

It is a most unfortunate fact that applications are not documented in a way that reaches the consciousness of researchers, teachers, students, and other application builders. There is a huge amount of experience “out there” that isn't documented or made accessible. This inevitably warps people's sense of reality — typically in the direction of what is new (or perceived as new) and described in trade press articles, academic papers, and textbooks. Much of the visible information is very open to fads and commercial manipulation. This leads to much “reinvention of the wheel”, suboptimal practice, and myths.

One common myth is that “most C++ code is just C code compiled with a C++ compiler”. There is nothing wrong with such code — after all, the C++ compiler will find more bugs than a C compiler — and such code is not uncommon. However, from seeing a lot of commercial C++ code and talking with innumerable developers and from talking with developers, I know that for many major applications, such as the ones mentioned here, the use of C++ is far more “adventurous”. It is common for developers to mention use of major locally designed class hierarchies, STL use, and use of “ideas from STL” in local code. See also §7.2.

Can we classify the application areas in which C++ is used? Here is one way of looking at it:

- Applications with systems components
- Banking and financial (funds transfer, financial modeling, customer interaction, teller machines, ...)
- Classical systems programming (compilers, operating systems, editors, database systems, ...)
- Conventional small business applications (inventory systems, customer service, ...)
- Embedded systems (instruments, cameras, cell phones, disc controllers, airplanes, rice cookers, medical systems, ...)
- Games
- GUI — iPod, CDE desktop, KDE desktop, Windows, ...
- Graphics
- Hardware design and verification [87]
- Low-level system components (device drivers, network layers, ...)
- Scientific and numeric computation (physics, engineering, simulations, data analysis, ...)
- Servers (web servers, large application backbones, billing systems, ...)
- Symbolic manipulation (geometric modeling, vision, speech recognition, ...)
- Telecommunication systems (phones, networking, monitoring, billing, operations systems, ...)

Again, this is more a list than a classification. The world of programming resists useful classification. However, looking at these lists, one thing should be obvious: C++ cannot be ideal for all of that. In fact, from the earliest days of C++, I have maintained that a general-purpose language can at most be second best in a well-defined application area and that C++ is “a general-purpose programming language with a bias towards systems programming”.

7.1 Applications Programming vs. Systems Programming

Consider an apparent paradox: C++ has a bias towards systems programming but “most programmers” write applications. Obviously millions of programmers are writing applications and many of those write their applications in C++. Why? Why don’t they write them in an applications programming language? For some definition of “applications programming language”, many do. We might define an “applications programming language” as one in which we can’t directly access hardware and that provides direct and specialized support for an important applications concept (such as data base access, user interaction, or numerical computation). Then, we can deem most languages “applications languages”: Excel, SQL, RPG, COBOL, Fortran, Java, C#, Perl, Python, etc. For good and bad, C++ is used as a general-purpose programming language in many areas where a more specialized (safer, easier to use, easier to optimize, etc.) language would seem applicable.

The reason is not just inertia or ignorance. I don’t claim that C++ is anywhere near perfect (that would be absurd) nor that it can’t be improved (we are working on C++0x, after all and see §9.4). However, C++ does have a niche — a very large niche — where other current languages fall short:

- applications with a significant systems programming component; often with resource constraints
- applications with components that fall into different application areas so that no single specialized applications language could support all

Application languages gain their advantages through specialization, through added conveniences, and through eliminating difficult to use or potentially dangerous features. Often, there is a run-time or space cost. Often, simplifications are based on strong assumptions about the execution environment. If you happen to need something fundamental that was deemed unnecessary in the design (such as direct access to memory or fully general abstraction mechanisms) or don’t need the “added conveniences” (and can’t afford the overhead they impose), C++ becomes a candidate. The basic conjecture on which C++ is built is that many applications have components for which that is the case: “Most of us do unusual things some of the time”.

The positive way of stating this is that general mechanisms beat special-purpose features for the majority of appli-

cations that does not completely fit into a particular classification, have to collaborate with other applications, or significantly evolve from their original narrow niche. This is one reason that every language seems to grow general-purpose features, whatever its original aims were and whatever its stated philosophy is.

If it was easy and cheap to switch back and forth among applications languages and general-purpose languages, we’d have more of a choice. However, that is rarely the case, especially where performance or machine-level access is needed. In particular, using C++ you can (but don’t have to) break some fundamental assumption on which an application language is built. The practical result is that if you need a systems programming or performance-critical facility of C++ somewhere in an application, it becomes convenient to use C++ for a large part of the application — and then C++’s higher-level (abstraction) facilities come to the rescue. C++ provides hardly any high-level features that are directly applicable in an application. What it offers are mechanisms for defining such facilities as libraries.

Please note that from a historical point of view this analysis need not be correct or the only possible explanation of the facts. Many prefer alternative ways of looking at the problem. Successful languages and companies have been built on alternative views. However, it is a fact that C++ was designed based on this view and that this view guided the evolution of C++; for example, see Chapter 9 of [121]. I consider it the reason that C++ initially succeeded in the mainstream and the reason that its use continued to grow steadily during the time period covered by this paper, despite the continuing presence of well-designed and better-financed alternatives in the marketplace. See also §9.4.

7.2 Programming Styles

C++ supports several programming styles or, as they are sometimes somewhat pretentiously called, “programming paradigms”. Despite that, C++ is often referred to as “an object-oriented programming language”. This is only true for some really warped definition of “object-oriented” and I never say just “C++ is an object-oriented language” [122]. Instead, I prefer “C++ supports object-oriented programming and other programming styles” or “C++ is a multi-paradigm programming language”. Programming style matters. Consequently, the way people refer to a language matters because it sets expectations and influences what people see as ideals.

C++ has C (C89) as an “almost subset” and supports the styles of programming commonly used for C [127]. Arguably, C++ supports those styles better than C does by providing more type checking and more notational support. Consequently, a lot of C++ code has been written in the style of C or — more commonly — in the style of C with a number of classes and class hierarchies thrown in without affecting the overall design. Such code is basically procedural, using classes to provide a richer set of types. That’s sometimes re-

ferred to as “C with Classes style”. That style can be significantly better (for understanding, debugging, maintenance, etc.) than pure C. However, it is less interesting from a historical perspective than code that also uses the C++ facilities to express more advanced programming techniques, and very often less effective than such alternatives. The fraction of C++ code written purely in “C style” appears to have been decreasing over the last 15 years (or more).

The abstract data type and object-oriented styles of C++ usage have been discussed often enough not to require explanation here (e.g., see [126]). They are the backbone of many C++ applications. However, there are limits to their utility. For example, object-oriented programming can lead to overly rigid hierarchies and overreliance on virtual functions. Also, a virtual function call is fundamentally efficient for cases where you need to select an action at run time, but it is still an indirect function call and thus expensive compared to an individual machine instruction. This has led to generic programming becoming the norm for C++ where high performance is essential (§6.1).

7.2.1 Generic programming

Sometimes, generic programming in C++ is defined as simply “using templates”. That’s at best an oversimplification. A better description from a programming language feature point of view is “parametric polymorphism” [107] plus overloading, which is selecting actions and constructing types based on parameters. A template is basically a compile-time mechanism for generating definitions (of classes and functions) based on type arguments, integer arguments, etc. [144].

Before templates, generic programming in C++ was done using macros [108], `void*`, and casts, or abstract classes. Naturally, some of that still persists in current use and occasionally these techniques have advantages (especially when combined with templated interfaces). However, the current dominant form of generic programming relies on class templates for defining types and function templates for defining operations (algorithms).

Being based on parameterization, generic programming is inherently more regular than object-oriented programming. One major conclusion from the years of use of major generic libraries, such as the STL, is that the current support for generic programming in C++ is insufficient. C++0x is taking care of at least part of that problem (§8).

Following Stepanov (§4.1.8), we can define generic programming without mentioning language features: Lift algorithms and data structures from concrete examples to their most general and abstract form. This typically implies representing the algorithms and their access to data as templates, as shown in the description of the STL (§4.1).

7.2.2 Template metaprogramming

The C++ template instantiation mechanism is (when compiler limits are ignored, as they usually can be) Turing com-

plete (e.g., see [150]). In the design of the template mechanism, I had aimed at full generality and flexibility. That generality was dramatically illustrated by Erwin Unruh in the early days of the standardization of templates. At an extensions working group meeting in 1994, he presented a program that calculated prime numbers at compile time (using error messages as the output mechanism) [143] and appeared surprised that I (and others) thought that marvelous rather than scary. Template instantiation is actually a small compile-time functional programming language. As early as 1995, Todd Veldhuizen showed how to define a compile-time if-statement using templates and how to use such if-statements (and switch-statements) to select among alternative data structures and algorithms [148]. Here is a compile-time if-statement with a simple use:

```
template<bool b, class X, class Y>
struct if_ {
    typedef X type; // use X if b is true
};

template<class X, class Y>
struct if_<

```

If the size of type `Foobar` is less than 40, the type of the variable `xy` is `Foo`; otherwise it is `Bar`. The second definition of `if_` is a partial specialization used when the template arguments match the `<false,X,Y>` pattern specified. It’s really quite simple, but very ingenious and I remember being amazed when Jeremy Siek first showed it to me. In a variety of guises, it has proven useful for producing portable high-performance libraries (e.g., most of the Boost libraries [16] rely on it).

Todd Veldhuizen also contributed the technique of expression templates [147], initially as part of the implementation of his high-performance numeric library Blitz++ [149]. The key idea is to achieve compile-time resolution and delayed evaluation by having an operator return a function object containing the arguments and operation to be (eventually) evaluated. The `<` operator generating a `Less_than` object in §4.1.4 is a trivial example. David Vandevoorde independently discovered this technique.

These techniques and others that exploit the computational power of template instantiation provide the foundation for techniques based on the idea of generating source code that exactly matches the needs of a given situation. It can lead to horrors of obscurity and long compile times, but also to elegant and very efficient solutions to hard problems; see [3, 2, 31]. Basically, template instantiation relies on over-

loading and specialization to provide a reasonably complete functional compile-time programming language.

There is no universally agreed-upon definition of the distinction between generic programming and template metaprogramming. However, generic programming tends to emphasize that each template argument type must have an enumerated well-specified set of properties; that is, it must be able to define a concept for each argument (§4.1.8, §8.3.3). Template metaprogramming doesn't always do that. For example, as in the `if_` example, template definitions can be chosen based on very limited aspects of an argument type, such as its size. Thus, the two styles of programming are not completely distinct. Template metaprogramming blends into generic programming as more and more requirements are placed on arguments. Often, the two styles are used in combination. For example, template metaprogramming can be used to select template definitions used in a generic part of a program.

When the focus of template use is very strongly on composition and selection among alternatives, the style of programming is sometimes called “generative programming” [31].

7.2.3 Multi-paradigm programming

It is important for programmers that the various programming styles supported by C++ are part of a single language. Often, the best code requires the use of more than one of the four basic “paradigms”. For example, we can write the classical “draw all shapes in a container” example from SIMULA BEGIN [9] like this:

```
void draw_all(vector<Shape*>& v)
{
    for_each(v.begin(), v.end(),           // sequence
              mem_fun(&Shape::draw)); // operation
}
```

Here, we use object-oriented programming to get the runtime polymorphism from the `Shape` class hierarchy. We use generic programming for the parameterized (standard library) container `vector` and the parameterized (standard library) algorithm `for_each`. We use ordinary procedural programming for the two functions `draw_all` and `mem_fun`. Finally, the result of the call of `mem_fun` is a function object, a class that is not part of a hierarchy and has no virtual functions, so that can be classified as abstract data type programming. Note that `vector`, `for_each`, `begin`, `end`, and `mem_fun` are templates, each of which will generate the most appropriate definition for its actual use.

We can generalize that to any sequence defined by a pair of `ForwardIterators`, rather than just `vectors` and improve type checking using C++0x concepts (§8.3.3):

```
template<ForwardIterator For>
void draw_all(For first, For last)
    requires SameType<For::value_type, Shape*>
{
```

```
    for_each(first, last, mem_fun(&Shape::draw));
}
```

I consider it a challenge to properly characterize multi-paradigm programming so that it can be easy enough to use for most mainstream programmers. This will involve finding a more descriptive name for it. Maybe it could even benefit from added language support, but that would be a task for C++1x.

7.3 Libraries, Toolkits, and Frameworks

So, what do we do when we hit an area in which the C++ language is obviously inadequate? The standard answer is: Build a library that supports the application concepts. C++ isn't an application language; it is a language with facilities supporting the design and implementation of elegant and efficient libraries. Much of the talk about object-oriented programming and generic programming comes down to building and using libraries. In addition to the standard library (§4) and the components from the library TR (§6.2), examples of widely used C++ libraries include

- ACE [95] — distributed computing
- Anti-Grain Geometry — 2D graphics
- Borland Builder (GUI builder)
- Blitz++[149] — vectors “The library that thinks it is a compiler”
- Boost[16] — foundation libraries building on the STL, graph algorithms, regular expression matching, threading, ...
- CGAL[23] — computational geometry
- Maya — 3D animation
- MacApp and PowerPlant — Apple foundation frameworks
- MFC — Microsoft Windows foundation framework
- Money++ — banking
- RogueWave library (pre-STL foundation library)
- STAPL[4], POOMA[86] — parallel computation
- Qt [10], FLTK [156], gtkmm [161], wxWigets [162] — GUI libraries and builders
- TAO [95], MICO, omniORB — CORBA ORBs
- VTK [155] — visualization

In this context, we use the word “toolkit” to describe a library supported by programming tools. In this sense, VTK is a toolkit because it contains tools for generating interfaces in Java and Python and Qt and FLTK are toolkits because they provide GUI builders. The combination of libraries and tools is an important alternative to dialects and special-purpose languages [133, 151].

A library (toolkit, framework) can support a huge user community. Such user communities can be larger than the

user communities of many programming languages and a library can completely dominate the world-view of their users. For example, Qt [10] is a commercial product with about 7,500 paying customers in 2006 plus about 150,000 users of its open-source version [141]. Two Norwegian programmers, Eirik Chambe-Eng and Haavard Nord, started what became Qt in 1991-92 and the first commercial release was in 1995. It is the basis of the popular desktop KDE (for Linux, Solaris, and FreeBSD) and well known commercial products, such as Adobe Photoshop Elements, Google Earth, and Skype (Voice over IP service).

Unfortunately for C++'s reputation, a good library cannot be seen; it just does its job invisibly to its users. This often leads people to underestimate the use of C++. However, “there are of course the Windows Foundation Classes (MFC), MacApp, and PowerPlant — most Mac and Windows commercial software is built with one of these frameworks” [85].

In addition to these general and domain-specific libraries, there are many much more specialized libraries. These have their function limited to a specific organization or application. That is, they apply libraries as an application design philosophy: “first extend the language by a library, then write the application in the resulting extended language”. The “string library” (part of a larger system called “Panther”) used by Celera Genomics as the base for their work to sequence the human genome [70] is a spectacular example of this approach. Panther is just one of many C++ libraries and applications in the general area of biology and biological engineering.

7.4 ABIs and Environments

Using libraries on a large scale isn't without problems. C++ supports source-level compatibility (but provides only weak link-time compatibility guarantees). That's fine if

- you have (all) the source
- your code compiles with your compiler
- the various parts of your source code are compatible (e.g., with respect to resource usage and error handling)
- your code is all in C++

For a large system, typically none of these conditions hold. In other words, linking is a can of worms. The root of this problem is the fundamental C++ design decision: Use existing linkers (D&E §4.5).

It is not guaranteed that two C translation units that match according to the language definition will link correctly when compiled with different compilers. However, for every platform, agreement has been reached for an ABI (Application Binary Interface) so that the register usage, calling conventions, and object layout match for all compilers so that C programs will correctly link. C++ compilers use these conventions for function call and simple structure layout. However, traditionally C++ compiler vendors have resisted link-

age standards for layout of class hierarchies, virtual function calls, and standard library components. The result is that to be sure that a legal C++ program actually works, every part (including all libraries) has to be compiled by the same compiler. On some platforms, notably Sun's and also Itanium (IA64) [60], C++ ABI standards exist but historically the rule “use a single compiler or communicate exclusively through C functions and structs” is the only really viable rule.

Sticking with one compiler can ensure link compatibility on a platform, but it can also be a valuable tool in providing portability across many platforms. By sticking to a single implementer, you gain “bug compatibility” and can target all platforms supported by that vendor. For Microsoft platforms, Microsoft C++ provides that opportunity; for a huge range of platforms, GNU C++ is portable; and for a diverse set of platforms, users get a pleasant surprise when they notice that many of their local implementations use an EDG (Edison Design Group) front-end making their source code portable. This only (sic!) leaves the problems of version skew. During this time period every C++ compiler went through a series of upgrades, partly to increase standard conformance, to adjust to platform ABIs, and to improve performance, debugging, integration with IDEs, etc.

Link compatibility with C caused a lot of problems, but also yielded significant advantages. Sean Parent (Adobe) observes: “one reason I see for C++'s success is that it is ‘close enough’ to C that platform vendors are able to provide a single C interface (such as the Win32 API or the Mac Carbon API) which is C++ compatible. Many libraries provide a C interface which is C++ compatible because the effort to do so is low — where providing an interface to a language such as Eiffel or Java would be a significant effort. This goes beyond just keeping the linking model the same as C but to the actual language compatibility”.

Obviously, people have tried many solutions to the linker problem. The platform ABIs are one solution. CORBA is a platform- and language-independent (or almost so) solution that has found widespread use. However, it seems that C++ and Java are the only languages heavily used with CORBA. COM was Microsoft's platform-dependent and language-independent solution (or almost so). One of the origins of Java was a perceived need to gain platform independence and compiler independence; the JVM solved that problem by eliminating language independence and completely specifying linkage. The Microsoft CLI (Common Language Infrastructure) solves the problem in a language-independent manner (sort of) by requiring all languages to support a Java-like linkage, metadata, and execution model. Basically all of these solutions provide platform independence by becoming a platform: To use a new machine or operating system, you port a JVM, an ORB, etc.

The C++ standard doesn't directly address the problem of platform incompatibilities. Using C++, platform indepen-

dence is provided through platform-specific code (typically relying on conditional compilation — `#ifdef`). This is often messy and ad hoc, but a high degree of platform independence can be provided by localizing dependencies in the implementation of a relatively simple platform layer — maybe just a single header file [16]. In any case, to implement the platform-independent services effectively, you need a language that can take advantage of the peculiarities of the various hardware and operating systems environments. More often than not, that language is C++.

Java, C#, and many other languages rely on metadata (that is, data that defines types and services associated with them) and provide services that depend on such metadata (such as marshalling of objects for transfer to other computers). Again, C++ takes a minimalist view. The only “metadata” available is the RTTI (§5.1.2), which provides just the name of the class and the list of its base classes. When RTTI was discussed, some of us dreamed of tools that would provide more data for systems that needed it, but such tools did not become common, general-purpose, or standard.

7.5 Tools and Research

Since the late 1980s, C++ developers have been supported by a host of analysis tools, development tools, and development environments available in the C++ world. Examples are:

- Visual Studio (IDE; Microsoft)
- KDE (Desktop Environment; Free Software)
- Xcode (IDE; Apple)
- lint++ (static analysis tool; Gimpel Software)
- Vtune (multi-level performance tuning; Intel)
- Shark (performance optimization; Apple)
- PreFAST (static source code analysis; Microsoft)
- Klocwork (static code analysis; Klocwork)
- LDRA (testing; LDRA Ltd.)
- QA.C++ (static analysis; Programming Research)
- Purify (memory leak finder; IBM Rational)
- Great Circle (garbage collector and memory usage analyzer; Geodesic, later Symantec)

However, tools and environments have always been a relative weakness of C++. The root of that problem is the difficulty of parsing C++. The grammar is not LR(N) for any N. That’s obviously absurd. The problem arose because C++ was based directly on C (I borrowed a YACC-based C parser from Steve Johnson), which was “known” not to be expressible as a LR(1) grammar until Tom Pennello discovered how to write one in 1985. Unfortunately, by then I had defined C++ in such a way that Pennello’s techniques could not be applied to C++ and there was already too much code dependent on the non-LR grammar to change C++. Another aspect

of the parsing problem is the macros in the C preprocessor. They ensure that what the programmer sees when looking at a line of code can — and often does — dramatically differ from what the compiler sees when parsing and type checking that same line of code. Finally, the builder of advanced tools must also face the complexities of name lookup, template instantiation, and overload resolution.

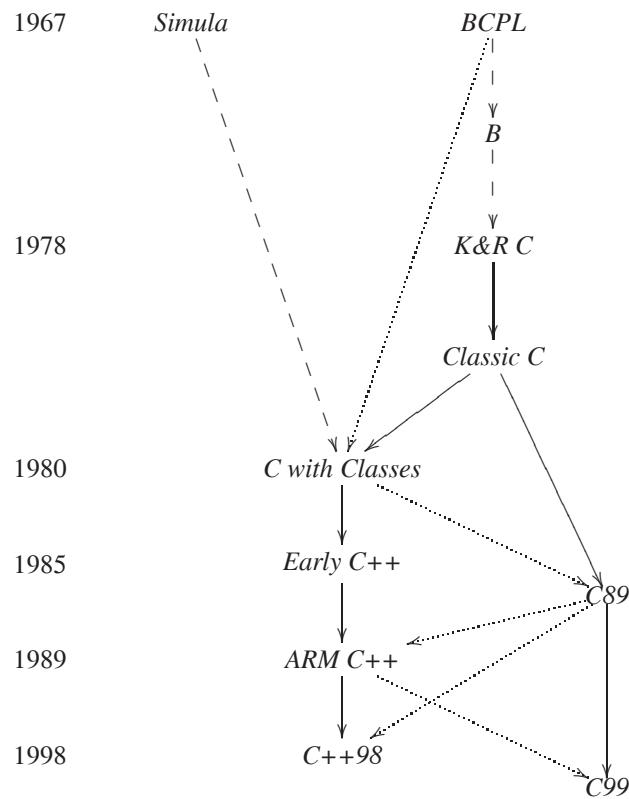
In combination, these complexities have confounded many attempts to build tools and programming environments for C++. The result was that far too few software development and source code analysis tools have become widely used. Also, the tools that were built tended to be expensive. This led to relatively fewer academic experiments than are conducted with languages that were easier to analyze. Unfortunately, many take the attitude that “if it isn’t standard, it doesn’t exist” or alternatively “if it costs money, it doesn’t exist”. This has led to lack of knowledge of and underuse of existing tools, leading to much frustration and waste of time.

Indirectly, this parsing problem has caused weaknesses in areas that rely on run-time information, such as GUI-builders. Since the language doesn’t require it, compilers generally don’t produce any form of metadata (beyond the minimum required by RTTI; §5.1.2). My view (as stated in the original RTTI papers and in D&E) was that tools could be used to produce the type information needed by a specific application or application area. Unfortunately, the parsing problem then gets in the way. The tool-to-generate-metadata approach has been successfully used for database access systems and GUI, but cost and complexity have kept this approach from becoming more widely used. In particular, academic research again suffered because a typical student (or professor) doesn’t have the time for serious infrastructure building.

A focus on performance also plays a part in lowering the number and range of tools. Like C, C++ is designed to ensure minimal run-time and space overheads. For example, the standard library `vector` is not by default range checked because you can build an optimal range-checked vector on top of an unchecked vector, but you cannot build an optimally fast vector on top of a range-checked one (at least not portably). However, many tools rely on additional actions (such as range-checking array access or validating pointers) or additional data (such as meta-data describing the layout of a data structure). A subculture of strong concern about performance came into the C++ community with much else from C. Often that has been a good thing, but it did have a limiting effect on tool building by emphasizing minimalism even where there were no serious performance issues. In particular, there is no reason why a C++ compiler couldn’t supply superb type information to tool builders [134].

Finally, C++ was a victim of its own success. Researchers had to compete with corporations that (sometimes correctly) thought that there was money to be made in the kind of tools researchers would like to build. There was also a curious

problem with performance: C++ was too efficient for any really significant gains to come easily from research. This led many researchers to migrate to languages with glaring inefficiencies for them to eliminate. Elimination of virtual function calls is an example: You can gain much better improvements for just about any object-oriented language than for C++. The reason is that C++ virtual function calls are very fast and that colloquial C++ already uses non-virtual functions for time-critical operations. Another example is garbage collection. Here the problem was that colloquial C++ programs don't generate much garbage and that the basic operations are fast. That makes the fixed overhead of a garbage collector looks far less impressive when expressed as a percentage of run time than it does for a language with less efficient basic operations and more garbage. Again, the net effect was to leave C++ poorer in terms of research and tools.



“Classic C” is what most people think of as K&R C, but C as defined in [76] lacks structure copy and enumerations. ARM C++ is C++ as defined by the ARM [35] and the basis for most pre-standard C++. The basic observation is that by now C (i.e., C89 or C99) and C++ (i.e., C++98) are siblings (with “Classic C” as their common ancestor), rather than the more conventional view that C++ is a dialect of C that somehow failed to be compatible. This is an important issue because people commonly proclaim the right of each language to evolve separately, yet just about everybody expects ISO C++ to adopt the features adopted by ISO C — despite the separate evolution of C and a tendency of the C committee to adopt features that are similar to but incompatible with what C++ already offers. Examples selected from a long list [127] are `bool`, `inline` functions, and `complex` numbers.

7.6 C/C++ Compatibility

C is C++’s closest relative and a high degree of C compatibility has always been a design aim for C++. In the early years, the primary reasons for compatibility were to share infrastructure and to guarantee completeness (§2.2). Later, compatibility remained important because of the huge overlap in applications areas and programmer communities. Many little changes were made to C++ during the ’80s and ’90s to bring C++ closer to ISO C [62] (C89). However, during 1995-2004, C also evolved. Unfortunately, C99 [64] is in significant ways less compatible with C++ than C89 [62] and harder to coexist with. See [127] for a detailed discussion of the C/C++ relationship. Here is a diagram of the relationships among the various generations of C and C++:

7.7 Java and Sun

I prefer not to compare C++ to other programming languages. For example, in the “Notes to the Reader” section of D&E [121], I write:

Several reviewers asked me to compare C++ to other languages. This I have decided against doing. ... Language comparisons are rarely meaningful and even less often fair. A good comparison of major programming languages requires more effort than most people are willing to spend, experience in a wide range of application areas, a rigid maintenance of a detached and impartial point of view, and a sense of fairness. I do not have the time, and as the designer of C++, my

impartiality would never be fully credible. ... Worse, when one language is significantly better known than others, a subtle shift in perspective occurs: Flaws in the well-known language are deemed minor and simple workarounds are presented, whereas similar flaws in other languages are deemed fundamental. Often, the workarounds commonly used in the less-well-known languages are simply unknown to the people doing the comparison or deemed unsatisfactory because they would be unworkable in the more familiar language. ... Thus, I restrict my comments about languages other than C++ to generalities and to very specific comments.

However, many claims about the C++/Java relationship have been made and the presence of Java in the marketplace has affected the C++ community. Consequently, a few comments are unavoidable even though a proper language comparison is far beyond the scope of this paper and even though Java has left no traces in the C++ definition.

Why not? From the earliest days of Java, the C++ committee has always included people with significant Java experience: users, implementers, tool builders, and JVM implementers. I think at a fundamental level Java and C++ are too different for easy transfer of ideas. In particular,

- C++ relies on direct access to hardware resources to achieve many of its goals whereas Java relies on a virtual machine to keep it away from the hardware.
- C++ is deliberately frugal with run-time support whereas Java relies on a significant amount of metadata
- C++ emphasizes interoperability with code written in other languages and sharing of system tools (such as linkers) whereas Java aims for simplicity by isolating Java code from other code.

The “genes” of C++ and Java are quite dissimilar. The syntactic similarities between Java and C++ have often been deceptive. As an analogy, I note that it is far easier for English to adopt “structural elements” from closely related languages, such as French or German, than from more different languages, such as Japanese or Thai.

Java burst onto the programming scene with an unprecedented amount of corporate backing and marketing (much aimed at non-programmers). According to key Sun people (such as Bill Joy), Java was an improved and simplified C++. “What Bjarne would have designed if he hadn’t had to be compatible with C” was — and amazingly still is — a frequently heard statement. Java is not that; for example, in D&E §9.2.2, I outlined fundamental design criteria for C++:

What would be a better language than C++ for the things C++ is meant for? Consider the first-order decisions:

- Use of static type checking and Simula-like classes.

- Clean separation between language and environment.
- C source compatibility (“as close as possible”).
- C link and layout compatibility (“genuine local variables”).
- No reliance on garbage collection.

I still consider static type checking essential for good design and run-time efficiency. Were I to design a new language for the kind of work done in C++ today, I would again follow the Simula model of type checking and inheritance, *not* the Smalltalk or Lisp models. As I have said many times, ‘Had I wanted an imitation Smalltalk, I would have built a much better imitation. Smalltalk is the best Smalltalk around. If you want Smalltalk, use it’.

I think I could express that more clearly today, but the essence would be the same; these criteria are what define C++ as a systems programming language and what I would be unwilling to give up. In the light of Java, that section seems more relevant today than when I wrote it in 1993 (pre-Java). Having the built-in data types and operators mapped directly to hardware facilities and being able to exploit essentially every machine resource is implicit in “C compatibility”.

C++ does not meet Java’s design criteria; it wasn’t meant to. Similarly, Java doesn’t meet C++’s design criteria. For example, consider a couple of language-technical criteria:

- Provide as good support for user-defined types as for built-in types
- Leave no room for a lower-level language below C++ (except assembler)

Many of the differences can be ascribed to the aim of keeping C++ a systems programming language with the ability to deal with hardware and systems at the lowest level and with the least overhead. Java’s stated aims seem more directed towards becoming an applications language (for some definition of “application”).

Unfortunately, the Java proponents and their marketing machines did not limit themselves to praising the virtues of Java, but stooped to bogus comparisons (e.g., [51]) and to name calling of languages seen as competitors (most notably C and C++). As late as 2001, I heard Bill Joy claim (orally with a slide to back it up in a supposedly technical presentation) that “reliable code cannot be written in C/C++ because they don’t have exceptions” (see §5, §5.3). I see Java as a Sun commercial weapon aimed at Microsoft that missed and hit an innocent bystander: the C++ community. It hurt many smaller language communities even more; consider Smalltalk, Lisp, Eiffel, etc.

Despite many promises, Java didn’t replace C++ (“Java will completely kill C++ within two years” was a graphic

expression I repeatedly heard in 1996). In fact, the C++ community has trebled in size since the first appearance of Java. The Java hype did, however, harm the C++ community by diverting energy and funding away from much-needed tools, library and techniques work. Another problem was that Java encouraged a limited “pure object-oriented” view of programming with a heavy emphasis on run-time resolution and a de-emphasis of the static type system (only in 2005 did Java introduce “generics”). This led many C++ programmers to write unnecessarily inelegant, unsafe, and poorly performing code in imitation. This problem gets especially serious when the limited view infects education and creates a fear of the unfamiliar in students.

As I predicted [136] when I first heard the boasts about Java’s simplicity and performance, Java rapidly accreted new features — in some cases paralleling C++’s earlier growth. New languages are always claimed to be “simple” and to become useful in a wider range of real-world applications they increase in size and complexity. Neither Java nor C++ was (or is) immune to that effect. Obviously, Java has made great strides in performance — given its initial slowness it couldn’t fail to — but the Java object model inhibits performance where abstraction is seriously used (§4.1.1, §4.1.4). Basically, C++ and Java are far more different in aims, language structure, and implementation model than most people seem to think. One way of viewing Java is as a somewhat restricted version of Smalltalk’s run-time model hidden behind a C++-like syntax and statically type-checked interfaces.

My guess is that Java’s real strength compared to C++ is the binary compatibility gained from Sun having de facto control of the linker as expressed through the definition of the Java virtual machine. This gives Java the link-compatibility that has eluded the C++ community because of the decision to use the basic system linkers and the lack of agreement among C++ vendors on key platforms (§7.4).

In the early 1990s, Sun developed a nice C++ compiler based on Mike Ball and Steve Clamage’s Taumetric compiler. With the release of Java and Sun’s loudly proclaimed pro-Java policy where C++ code was referred to (in advertising “literature” and elsewhere) as “legacy code” that needed rewriting and as “contamination”, the C++ group suffered some lean years. However, Sun never wavered in its support of the C++ standards efforts and Mike, Steve and others made significant contributions. Technical people have to live with technical realities — such as the fact that many Sun customers and many Sun projects rely on C++ (in total or in part). In particular, Sun’s Java implementation, HotSpot, is a C++ program.

7.8 Microsoft and .Net

Microsoft is currently the 800-pound gorilla of software development and it has had a somewhat varied relationship with C++. Their first attempt at an object-oriented C in the late 1980s wasn’t C++ and I have the impression that a cer-

tain ambivalence about standard conformance lingers. Microsoft is better known for setting de facto standards than for strictly sticking to formal ones. However, they did produce a C++ compiler fairly early. Its main designer was Martin O’Riordan, who came from the Irish company Glockenspiel where he had been a Cfront expert and produced and maintained many ports. He once amused himself and his friends by producing a Cfront that spoke its error messages through a voice synthesizer in (what he believed to be) a thick Danish accent. To this day there are ex-Glockenspiel Irishmen on the Microsoft C++ team.

Unfortunately, that first release didn’t support templates or exceptions. Eventually, those key features were supported and supported well, but that took years. The first Microsoft compiler to provide a close-to-complete set of ISO C++ features was VC++ 6.0, released in July 1998; its predecessor, VC++ 5.0 from February 1997, already had many of the key features. Before that, some Microsoft managers used highly visible platforms, such as conference keynotes, for some defensive bashing of these features (as provided only by their competitors, notably Borland) as “expensive and useless”. Worse, internal Microsoft projects (which set widely followed standards) couldn’t use templates and exceptions because their compiler didn’t support those features. This established bad programming practices and did long-term harm.

Apart from that, Microsoft was a responsible and attentive member of the community. Microsoft sent and sends members to the committee meetings and by now — somewhat belatedly — provides an excellent C++ compiler with good standard conformance.

To effectively use the .Net framework, which Microsoft presents as the future of Windows, a language has to support a Java-like set of facilities. This implies support for a large number of language features including a massive metadata mechanism and inheritance — complete with covariant arrays (§4.1.1). In particular, a language used to produce components for consumption by other languages must produce complete metadata and a language that wants to consume components produced by other languages must be able to accept the metadata they produce. The Microsoft C++ dialect that supports all that, ISO C++ plus “The CLI extensions to ISO C++”, colloquially referred to as C++/CLI [41], will obviously play a major role in the future of the C++ world. Interestingly, C++ with the C++/CLI extensions is the only language that provides access to every feature of .Net. Basically, C++/CLI is a massive set of extensions to ISO C++ and provides a degree of integration with Windows that makes it unlikely that a program that relies on C++/CLI features in any significant way will be portable beyond the Microsoft platforms that provide the massive infrastructure on which C++/CLI depends. As ever, systems interfaces can be encapsulated, and must be encapsulated to preserve portability. In addition to ISO C++, C++/CLI provides its own loop

construct, overloading mechanisms (indexers), “properties”, event mechanism, garbage-collected heap, a different class object initialization semantics, a new form of references, a new form of pointers, generics, a new set of standard containers (the .Net ones), and more.

In the early years of .Net (around 2000 onwards), Microsoft provided a dialect called “managed C++” [24]. It was generally considered “pretty lame” (if essential to some Microsoft users) and appears to have been mostly a stop-gap measure — without any clear indication what the future might bring for its users. It has now been superseded by the much more comprehensive and carefully designed C++/CLI.

One of my major goals in working in the standards committee was to prevent C++ from fracturing into dialects. Clearly, in the case of C++/CLI, I and the committee failed. C++/CLI has been standardized by ECMA [41] as a binding to C++. However, Microsoft’s main products, and those of their major customers, are and will remain in C++. This ensures good compiler and tool support for C++ — that is ISO C++ — under Windows for the foreseeable future. People who care about portability can program around the Microsoft extensions to ensure platform independence (§7.4). By default the Microsoft compiler warns about use of C++/CLI extensions.

The members of the C++ standards committee were happy to see C++/CLI as an ECMA standard. However, an attempt to promote that standard to an ISO standard caused a flood of dissent. Some national standards bodies — notably the UK’s C++ Panel — publicly expressed serious concern [142]. This caused people at Microsoft to better document their design rationale [140] and to be more careful about not confusing ISO C++ and C++/CLI in Microsoft documentation.

7.9 Dialects

Obviously, not everyone who thought C++ to be basically a good idea wanted to go through the long and often frustrating ISO standards process to get their ideas into the mainstream. Similarly, some people consider compatibility overrated or even a very bad idea. In either case, people felt they could make faster progress and/or better by simply defining and implementing their own dialect. Some hoped that their dialect would eventually become part of the mainstream; others thought that life outside the mainstream was good enough for their purposes, and a few genuinely aimed at producing a short-lived language for experimental purposes.

There have been too many C++ dialects (many dozens) for me to mention more than a tiny fraction. This is not ill will — though I am no great fan of dialects because they fracture the user community [127, 133] — but a reflection that they have had very little impact outside limited communities. So far, no major language feature has been brought into the mainstream of C++ from a dialect. However, C++0x will get something that looks like C++/CLI’s properly scoped enums [138], a C++/CLI-like for-statement

[84], and the keyword `nullptr` [139] (which curiously enough was suggested by me for C++/CLI).

Concurrency seems to be an extremely popular area of language extension. Here are a few C++ dialects supporting some form of concurrency with language features:

- Concurrent C++[46]
- micro C++[18]
- ABC++ [83]
- Charm++ [75]
- POOMA [86]
- C++// [21]

Basically, every trend and fad in the computer science world spawned a couple of C++ dialects, such as Aspect C++ [100], R++ [82], Compositional C++ [25], Objective C++ [90], Open C++ [26], and dozens more. The main problem with these dialects is their number. Dialects tend to split up a sub-user community to the point where none reach a large enough user community to afford a sustainable infrastructure [133].

Another kind of dialect appears when a vendor adds some (typically minor) “neat features” to their compiler to help their users with some specific tasks (e.g. OS access and optimization). These become barriers to portability even if they are often beloved by some users. They are also appreciated by some managers and platform fanatics as a lock-in mechanism. Examples exist for essentially every implementation supplier; for example Borland (Delphi-style properties), GNU (variable and type attributes), and Microsoft (C++/CLI; §7.8). Such dialect features are nasty when they appear in header files, setting off a competitive scramble among vendors keen on being able to cope with their competitors’ code. They also significantly complicate compiler, library, and tool building.

It is not obvious when a dialect becomes a completely separate language and many languages borrowed heavily from C++ (with or without acknowledgments) without aiming for any degree of compatibility. A recent example is the “D” language (the currently most recent language with that popular name): “D was conceived in December 1999 by Walter Bright as a reengineering of C and C++” [17].

Even Java started out (very briefly) as a C++ dialect [164], but its designers soon decided that maintaining compatibility with C++ would be too constraining for their needs. Since their aims included 100% portability of Java code and a restriction of code styles to a version of object-oriented programming, they were almost certainly correct in that. Achieving any degree of useful compatibility is very hard, as the C/C++ experience shows.

8. C++0x

From late 1997 until 2002, the standards committee deliberately avoided serious discussion of language extension. This

allowed compiler, tool, and library implementers to catch up and users to absorb the programming techniques supported by Standard C++. I first used the term “C++0x” in 2000 and started a discussion of “directions for C++0x” through presentations in the committee and elsewhere from 2001 onwards. By 2003, the committee again considered language extensions. The “extensions working group” was reconstituted as the “evolution working group”. The name change (suggested by Tom Plum) is meant to reflect a greater emphasis on the integration of language features and standard library facilities. As ever, I am the chairman of that working group, hoping to help ensure a continuity of vision for C++ and a coherence of the final result. Similarly, the committee membership shows continuous participation of a large number of people and organizations. Fortunately, there are also many new faces bringing new interests and new expertise to the committee.

Obviously, C++0x is still work in progress, but years of work are behind us and many votes have been taken. These votes are important in that they represent the response of an experienced part of the C++ community to the problems with C++98 and the current challenges facing C++ programmers. The committee intended to be cautious and conservative about changes to the language itself, and strongly emphasize compatibility. The stated aim was to channel the major effort into an expansion of the standard library. In the standard library, the aim was to be aggressive and opportunistic. It is proving difficult to deliver on that aim, though. As usual, the committee just doesn’t have sufficient resources. Also, people seem to get more excited over language extensions and are willing to spend more time lobbying for and against those.

The rate of progress is roughly proportional to the number of meetings. Since the completion of the 1998 standard, there has been two meetings a year, supplemented by a large amount of web traffic. As the deadlines for C++0x approach, these large meetings are being supplemented by several “extra” meetings focused on pressing topics, such as concepts (§8.3.3) and concurrency (§5.5).

8.1 Technical Challenges

What technical challenges faced the C++ community at the time when C++0x was beginning to be conceived? At a high level an answer could be:

- GUI-based application building
- Distributed computing (especially the web)
- Security

The big question is how to translate that into language features, libraries (ISO standard or not), programming environment, and tools. In 2000-2002, I tried unsuccessfully to get the standards committee’s attention on the topic of distributed computing and Microsoft representatives regularly try to raise the topic of security. However, the committee

simply doesn’t think like that. To make progress, issues have to be expressed as concrete proposals for change, such as to add a (specific) language feature to support callbacks or to replace the notoriously unsafe C standard library function `gets()` with a (specific) alternative. Also, the C++ tradition is to approach challenges obliquely through improved abstraction mechanisms and libraries rather than providing a solution for a specific problem.

After I had given a couple of talks on principles and directions, the first concrete action in the committee was a brainstorming session at the 2001 meeting in Redmond, Washington. Here we made a “wish list” for features for C++0x. The first suggestion was for a portable way of expressing alignment constraints (by P. J. Plauger supported by several other members). Despite my strongly expressed urge to focus on libraries, about 90 out of about 100 suggestions were for language features. The saving grace was that many of those were for features that would ease the design, implementation, and use of more elegant, more portable, and efficient libraries. This brainstorming provided the seeds of maintained “wish lists” for C++0x language features and standard library facilities [136].

From many discussions and presentations, I have come with a brief summary of general aims and design rules for C++0x that appear to be widely accepted. Aims:

- Make C++ a better language for systems programming and library building — rather than providing specialized facilities for a particular sub-community (e.g. numeric computation or Windows-style application development)
- Make C++ easier to teach and learn — through increased uniformity, stronger guarantees, and facilities supportive of novices (there will always be more novices than experts)

Rules of thumb:

- Provide stability and compatibility
- Prefer libraries to language extensions
- Make only changes that change the way people think
- Prefer generality to specialization
- Support both experts and novices
- Increase type safety
- Improve performance and ability to work directly with hardware
- Fit into the real world

These lists have been useful as a framework for rationales for proposed extensions. Dozens of committee technical papers have used them. However, they provide only a weak set of directions and are only a weak counterweight to a widespread tendency to focus on details. The GUI and distributed computing challenges are not directly represented here, but fea-

ture prominently under “libraries” and “work directly with hardware” (§8.6).

8.2 Suggested Language Extensions

To give a flavor of the committee work around 2005, consider a (short!) incomplete list of proposed extensions:

1. `decltype` and `auto` — type deduction from expressions (§8.3.2)
2. Template aliases — a way of binding some but not all template parameters and naming the resulting partial template specialization
3. Extern templates — a way of suppressing implicit instantiation in a translation unit
4. Move semantics (rvalue references) — a general mechanism for eliminating redundant copying of values
5. Static assertions (`static_assert`)
6. `long long` and many other C99 features
7. `>>` (without a space) to terminate two template specializations
8. Unicode data types
9. Variadic templates (§8.3.1)
10. Concepts — a type system for C++ types and integer values (§8.3.3)
11. Generalized constant expressions [39]
12. Initializer lists as expressions (§8.3.1)
13. Scoped and strongly typed enumerations [138]
14. Control of alignment
15. `nullptr` — Null pointer constant [139]
16. A `for`-statement for ranges
17. Delegating constructors
18. Inherited constructors
19. Atomic operations
20. Thread-local storage
21. Defaulting and inhibiting common operations
22. Lambda functions
23. Programmer-controlled garbage collection [12] (§5.4)
24. In-class member initializers
25. Allow local classes as template parameters
26. Modules
27. Dynamic library support
28. Integer sub-ranges
29. Multi-methods
30. Class namespaces
31. Continuations

32. Contract programming — direct support for preconditions, postconditions, and more
33. User-defined operator. (dot)
34. `switch` on string
35. Simple compile-time reflection
36. `#nomacro` — a kind of scope to protect code from unintended macro expansion
37. GUI support (e.g., slots and signals)
38. Reflection (i.e., data structures describing types for runtime use)
39. concurrency primitives in the language (not in a library)

As ever, there are far more proposals than the committee could handle or the language could absorb. As ever, even accepting all the good proposals is infeasible. As ever, there seems to be as many people claiming that the committee is spoiling the language by gratuitous complicated features as there are people who complain that the committee is killing the language by refusing to accept essential features. If you take away consistent overstatement of arguments, both sides have a fair degree of reason behind them. The balancing act facing the committee is distinctly nontrivial.

As of October 2008, Items 1-7 have been approved. Based on the state of proposals and preliminary working group votes, my guess is that items 10-21 will also be accepted. Beyond that, it's hard to guess. Proposals 22-25 are being developed aiming for votes in July 2007 and proposal 26 (modules) has been postponed to a technical report.

Most of these are being worked upon under one or more of the “rules of thumb” listed above. That list is less than half of the suggestions that the committee has received in a form that compels it to (at least briefly) consider them. My collection of suggestions from emails and meetings with users is several times that size. At my urging, the committee at the spring 2005 meeting in Lillehammer, Norway decided (by vote) to stop accepting new proposals. In October of 2006, this vote was followed up by a vote to submit a draft standard in late 2007 so as to make C++0x into C++09. However, even with the stream of new proposals stemmed, it is obvious that making a coherent whole of a selection of features will be a practical challenge as well as a technical one.

To give an idea of the magnitude of the technical challenge, consider that a paper on part of the concepts problem was accepted for the premier academic conference in the field, POPL, in 2006 [38] and other papers analyzing problems related to concepts were presented at OOPSLA [44, 52] and PLDI [74]. I personally consider the technical problems related to the support of concurrency (including the memory model) harder still — and essential. The C++0x facilities for dealing with concurrency are briefly discussed in §8.6.

The practical challenge is to provide correct and consistent detailed specifications of all these features (and stan-

dard library facilities). This involves producing and checking hundreds of pages of highly technical standards text. More often than not, implementation and experimentation are part of the effort to ensure that a feature is properly specified and interacts well with other features in the language and the standard library. The standard specifies not just an implementation (like a vendor's documentation), but a whole set of possible implementations (different vendors will provide different implementations of a feature, each with the same semantics, but with different engineering tradeoffs). This implies an element of risk in accepting any new feature — even in accepting a feature that has already been implemented and used. Committee members differ in their perception of risk, in their views of what risks are worth taking, and in their views of how risk is best handled. Naturally, this is a source of many difficult discussions.

All proposals and all the meeting minutes of the committee are available on the committee's website [69]. That's more than 2000 documents — some long. A few of the pre-1995 papers are not yet (August 2006) available online because the committee relied on paper until about 1994.

8.3 Support for Generic Programming

For the language itself, we see an emphasis on features to support generic programming because generic programming is the area where use of C++ has progressed the furthest relative to the support offered by the language.

The overall aim of the language extensions supporting generic programming is to provide greater uniformity of facilities so as to make it possible to express a larger class of problems directly in a generic form. The potentially most significant extensions of this kind are:

- general initializer lists (§8.3.1)
- auto (§8.3.2)
- concepts (§8.3.3)

Of these proposals, only `auto` has been formally voted in. The others are mature proposals and initial “straw votes” have been taken in their support.

8.3.1 General initializer lists

“Provide as good support for user-defined types as for built-in types” is prominent among the language-technical design principles of C++ (§2). C++98 left a major violation of that principle untouched: C++98 provides notational support for initializing a (built-in) array with a list of values, but there is no such support for a (user-defined) vector. For example, we can easily define an array initialized to the three ints 1, 2, and 3:

```
int a[] = { 1,2,3 };
```

Defining an equivalent vector is awkward and may require the introduction of an array:

```
// one way:  
vector v1;  
v1.push_back(1);  
v1.push_back(2);  
v1.push_back(3);  
  
// another way  
int a[] = { 1,2,3 };  
vector v2(a,a+sizeof(a)/sizeof(int));
```

In C++0x, this problem will be remedied by allowing a user to define a “sequence constructor” for a type to define what initialization with an initializer list means. By adding a sequence constructor to `vector`, we would allow:

```
vector<int> v = { 1,2,3 };
```

Because the semantics of initialization defines the semantics of argument passing, this will also allow:

```
int f(const vector<int>&);  
// ...  
int x = f({ 1,2,3 });  
int y = f({ 3,4,5,6,7,8, 9, 10 });
```

That is, C++0x gets a type-safe mechanism for variable-length homogeneous argument lists [135].

In addition, C++0x will support type-safe variadic template arguments [54] [55]. For example:

```
template<class ... T> void print(const T& ...);  
// ...  
string name = "World"  
print("Hello, ",name,'!');  
int x = 7;  
print("x = ",x);
```

At the cost of generating a unique instantiation (from a single template function) for each call with a given set of argument types, variadic templates allow arbitrary non-homogenous argument lists to be handled. This is especially useful for tuple types and similar abstractions that inherently deal with heterogeneous lists.

I was the main proponent of the homogenous initializer list mechanism. Doug Gregor and Jaakko Järvi and Doug Gregor designed the variadic template mechanism, which conceptually has its roots in Jakko Järvi and Gary Powell's lambda library [72] and the tuple library [71].

8.3.2 Auto

C++0x will support the notion of a variable being given a type deduced from the type of its initializer [73]. For example, we could write the verbose example from §4.1.3 like this:

```
auto q = find(vi.begin(),vi.end(),7); // ok
```

Here, we deduce the type of `q` to be the return type of the value returned by `find`, which in turn is the type of `vi.begin()`; that is, `vector<int>::iterator`. I first implemented that use of `auto` in 1982, but was forced to back

it out of “C with Classes” because of a C compatibility problem. In K&R C [76] (and later in C89 and ARM C++), we can omit the type in a declaration. For example:

```
static x;      // means static int x
auto y;       // means stack allocated int y
```

After a proposal by Dag Brućk, both C99 and C++98 banned this “implicit int”. Since there is now no legal code for “auto q” to be incompatible with, we allowed it. That incompatibility was always more theoretical than real (reviews of huge amounts of code confirm that), but using the (unused) keyword auto saved us from introducing a new keyword. The obvious choice (from many languages, including BCPL) is let, but every short meaningful word has already been used in many programs and long and cryptic keywords are widely disliked.

If the committee — as planned — accepts overloading based on concepts (§8.3.3) and adjusts the standard library to take advantage, we can even write:

```
auto q = find(vi,7); // ok
```

in addition to the more general, but wordier:

```
auto q = find(vi.begin(),vi.end(),7); // ok
```

Unsurprisingly, given its exceptionally long history, I was the main designer of the auto mechanism. As with every C++0x proposal, many people contributed, notably Gabriel Dos Reis, Jaakko Järvi, and Walter Brown.

8.3.3 Concepts

The D&E [121] discussion of templates contains three whole pages (§15.4) on constraints on template arguments. Clearly, I felt the need for a better solution — and so did many others. The error messages that come from slight errors in the use of a template, such as a standard library algorithm, can be spectacularly long and unhelpful. The problem is that the template code’s expectations of its template arguments are implicit. Consider again `find_if`:

```
template<class In, class Pred>
In find_if(In first, In last, Pred pred)
{
    while (first!=last && !pred(*first))
        ++first;
    return first;
}
```

Here, we make a lot of assumptions about the `In` and `Pred` types. From the code, we see that `In` must somehow support `!=`, `*`, and `++` with suitable semantics and that we must be able to copy `In` objects as arguments and return values. Similarly, we see that we can call a `Pred` with an argument of whichever type `*` returns from an `In` and apply `!` to the result to get something that can be treated as a `bool`. However, that’s all implicit in the code. Note that a lot of the flexibility of this style of generic programming comes from implicit conversions used to make template argument types

meet those requirements. The standard library carefully documents these requirements for input iterators (our `In`) and predicates (our `Pred`), but compilers don’t read manuals. Try this error and see what your compiler says:

```
find_if(1,5,3.14);           // errors
```

On 2000-vintage C++ compilers, the resulting error messages were uniformly spectacularly bad.

Constraints classes A partial, but often quite effective, solution based on my old idea of letting a constructor check assumptions about template arguments (D&E §15.4.2) is now finding widespread use. For example:

```
template<class T> struct Forward_iterator {
    static void constraints(T a) {
        ++a; a++;           // can increment
        T b = a; b = a;     // can copy
        *b = *a;            // can dereference
                            // and copy result
    }
    Forward_iterator()
    { void (*p)(T) = constraints; }
};
```

This defines a class that will compile if and only if `T` is a forward iterator [128]. However, a `Forward_iterator` object doesn’t really do anything, so that a compiler can (and all do) trivially optimize away such objects. We can use `Forward_iterator` in a definition like this:

```
template<class In, class Pred>
In find_if(In first, In last, Pred pred)
{
    Forward_iterator<In>(); // check
    while (first!=last && !pred(*first))
        ++first;
    return first;
}
```

Alex Stepanov and Jeremy Siek did a lot to develop and popularize such techniques. One place where they are used prominently is in the Boost library [16]. The difference in the quality of error messages can be spectacular. However, it is not easy to write constraints classes that consistently give good error messages on all compilers.

Concepts as a language feature Constraints classes are at best a partial solution. In particular, the type checking is done in the template definition. For proper separation of concerns, checking should rely only on information presented in a declaration. That way, we would obey the usual rules for interfaces and could start considering the possibility of genuine separate compilation of templates.

So, let’s tell the compiler what we expect from a template argument:

```
template<ForwardIterator In, Predicate Pred>
In find_if(In first, In last, Pred pred);
```

Assuming that we can express what a `ForwardIterator` and a `Predicate` are, the compiler can now check a call of `find_if` in isolation from its definition. What we are doing here is to build a type system for template arguments. In the context of modern C++, such “types of types” are called *concepts* (see §4.1.8). There are various ways of specifying such concepts; for now, think of them as a kind of constraints classes with direct language support and a nicer syntax. A concept says what facilities a type must provide, but nothing about how it provides those facilities. The use of a concept as the type of a template argument (e.g. `<ForwardIterator In>`) is very close to a mathematical specification “for all types `In` such that an `In` can be incremented, dereferenced, and copied”, just as the original `<class T>` is the mathematical “for all types `T`”.

Given only that declaration (and not the definition) of `find_if`, we can write

```
int x = find_if(1,2,Less_than<int>(7));
```

This call will fail because `int` doesn’t support unary `*` (dereference). In other words, the call will fail to compile because an `int` isn’t a `ForwardIterator`. Importantly, that makes it easy for a compiler to report the error in the language of the user and at the point in the compilation where the call is first seen.

Unfortunately, knowing that the iterator arguments are `ForwardIterators` and that the predicate argument is a `Predicate` isn’t enough to guarantee successful compilation of a call of `find_if`. The two argument types interact. In particular, the predicate takes an argument that is an iterator dereferenced by `*: pred(*first)`. Our aim is complete checking of a template in isolation from the calls and complete checking of each call without looking at the template definition. So, “concepts” must be made sufficiently expressive to deal with such interactions among template arguments. One way is to parameterize the concepts in parallel to the way the templates themselves are parameterized. For example:

```
template<Value_type T,
        ForwardIterator<T> In, // sequence of Ts
        Predicate<bool,T> Pred> // takes a T;
                                // returns a bool
In find_if(In first, In last, Pred pred);
```

Here, we require that the `ForwardIterator` must point to elements of a type `T`, which is the same type as the `Predicate`’s argument type. However, that leads to overly rigid interactions among template argument types and very complex patterns and redundant of parameterization [74]. The current concept proposal [129, 130, 132, 97, 53] focuses on expressing relationships among arguments directly:

```
template<ForwardIterator In, // a sequence
        Predicate Pred> // returns a bool
        requires Callable<Pred,In::value_type>
In find_if(In first, In last, Pred pred);
```

Here, we require that the `In` must be a `ForwardIterator` with a `value_type` that is acceptable as an argument to `Pred` which must be a `Predicate`.

A *concept* is a compile-time predicate on a set of types and integer values. The concepts of a single type argument provide a type system for C++ types (both built-in and user-defined types) [132].

Specifying template’s requirements on its argument using concepts also allows the compiler to catch errors in the template definition itself. Consider this plausible pre-concept definition:

```
template<class In, class Pred>
In find_if(In first, In last, Pred pred)
{
    while(first!=last && !pred(*first))
        first = first+1;
    return first;
}
```

Test this with a vector or an array, and it will work. However, we specified that `find_if` should work for a `ForwardIterator`. That is, `find_if` should be usable for the kind of iterator that we can supply for a list or an input stream. Such iterators cannot simply move `N` elements forward (by saying `p=p+N`) — not even for `N==1`. We should have said `++first`, which is not just simpler, but correct. Experience shows that this kind of error is very hard to catch, but concepts make it trivial for the compiler to detect it:

```
template<ForwardIterator In, Predicate Pred>
    requires Callable<Pred,In::value_type>
In find_if(In first, In last, Pred pred)
{
    while(first!=last && !pred(*first))
        first = first+1;
    return first;
}
```

The `+` operator simply isn’t among the operators specified for a `ForwardIterator`. The compiler has no problems detecting that and reporting it succinctly.

One important effect of giving the compiler information about template arguments is that overloading based on the properties of argument types becomes easy. Consider again `find_if`. Programmers often complain about having to specify the beginning and the end of a sequence when all they want to do is to find something in a container. On the other hand, for generality, we need to be able to express algorithms in terms of sequences delimited by iterators. The obvious solution is to provide both versions:

```
template<ForwardIterator In, Predicate Pred>
    requires Callable<Pred,In::value_type>
In find_if(In first, In last, Pred pred);

template<Container C, Predicate Pred>
    requires Callable<Pred,C::value_type>
In find_if(C& c, Pred pred);
```

Given that, we can handle examples like the one in §8.3.2 as well as examples that rely on more subtle differences in the argument types.

This is not the place to present the details of the concept design. However, as presented above, the design appears to have a fatal rigidity: The expression of required properties of a type is often in terms of required member types. However, built-in types do not have members. For example, how can an `int*` be a `ForwardIterator` when a `ForwardIterator` as used above is supposed to have a member `value_type`? In general, how can we write algorithms with precise and detailed requirements on argument types and expect that authors of types will define their types with the required properties? We can't. Nor can we expect programmers to alter old types whenever they find a new use for them. Real-world types are often defined in ignorance of their full range of uses. Therefore, they often fail to meet the detailed requirements even when they possess all the fundamental properties needed by a user. In particular, `int*` was defined 30 years ago without any thought of C++ or the STL notion of an iterator. The solution to such problems is to (non-intrusively) map the properties of a type into the requirements of a concept. In particular, we need to say “when we use a pointer as a forward iterator we should consider the type of the object pointed to its `value_type`”. In the C++0x syntax, that is expressed like this:

```
template<class T>
concept_map ForwardIterator<T*> {
    typedef T value_type;
}
```

A `concept_map` provides a map from a type (or a set of types; here, `T*`) to a concept (here, `ForwardIterator`) so that users of the concept for an argument will see not the actual type, but the mapped type. Now, if we use `int*` as a `ForwardIterator` that `ForwardIterator`'s `value_type` will be `int`. In addition to providing the appearance of members, a `concept_map` can be used to provide new names for functions, and even to provide new operations on objects of a type.

Alex Stepanov was probably the first to call for “concepts” as a C++ language feature [104] (in 2002) — I don't count the numerous vague and nonspecific wishes for “better type checking of templates”. Initially, I was not keen on the language approach because I feared it would lead in the direction of rigid interfaces (inhibiting composition of separately developed code and seriously hurting performance), as in earlier ideas for language-supported “constrained genericity”. In the summer of 2003, Gabriel Dos Reis and I analyzed the problem, outlined the design ideals, and documented the basic approaches to a solution [129] [130]. So, the current design avoids those ill effects (e.g., see [38]) and there are now many people involved in the design of concepts for C++0x, notably Doug Gregor, Jaakko Järvi, Gabriel Dos Reis, Jeremy Siek, and me. An experimental implemen-

tation of concepts has been written by Doug Gregor, together with a version of the STL using concepts [52].

I expect concepts to become central to all generic programming in C++. They are already central to much design using templates. However, existing code — not using concepts — will of course continue to work.

8.4 Embarrassments

My other priority (together with better support for generic programming) is better support for beginners. There is a remarkable tendency for proposals to favor the expert users who propose and evaluate them. Something simple that helps only novices for a few months until they become experts is often ignored. I think that's a potentially fatal design bias. Unless novices are sufficiently supported, only few will become experts. Bootstrapping is most important! Further, many — quite reasonably — don't want to become experts; they are and want to remain “occasional C++ users”. For example, a physicist using C++ for physics calculations or the control of experimental equipment is usually quite happy being a physicist and has only limited time for learning programming techniques. As computer scientists we might wish for people to spend more time on programming techniques, but rather than just hoping, we should work on removing unnecessary barriers to adoption of good techniques. Naturally, most of the changes needed to remove “embarrassments” are trivial. However, their solution is typically constrained by compatibility concerns and concerns for the uniformity of language rules.

A very simple example is

```
vector<vector<double>> v;
```

In C++98, this is a syntax error because `>>` is a single lexical token, rather than two `>`s each closing a template argument list. A correct declaration of `v` would be:

```
vector< vector<double> > v;
```

I consider that an embarrassment. There are perfectly good language-technical reasons for the current rule and the extensions working group twice rejected my suggestions that this was a problem worth solving. However, such reasons are language technical and of no interest to novices (of all backgrounds — including experts in other languages). Not accepting the first and most obvious declaration of `v` wastes time for users and teachers. A simple solution of the “`>>` problem” was proposed by David Vandevoorde [145] and voted in at the 2005 Lillehammer meeting, and I expect many small “embarrassments” to be absent from C++0x. However, my attempt together with Francis Glassborow and others, to try to systematically eliminate the most frequently occurring such “embarrassments” seems to be going nowhere.

Another example of an “embarrassment” is that it is legal to copy an object of a class with a user-defined destructor using a default copy operation (constructor or assignment).

Requiring user-defined copy operations in that case would eliminate a lot of nasty errors related to resource management. For example, consider an oversimplified string class:

```
class String {
public:
    String(char* pp);           // constructor
    ~String() { delete[] pp; }   // destructor
    char& operator[](int i);
private:
    int sz;
    char* p;
};

void f(char* x)
{
    String s1(x);
    String s2 = s1;
}
```

After the construction of `s2`, `s1.p` and `s2.p` point to the same memory. This memory (allocated by the constructor) will be deleted twice by the destructor, probably with disastrous results. This problem is obvious to the experienced C++ programmer, who will provide proper copy operations or prohibit copying. The problem has also been well documented from the earliest days of C++: The two obvious solutions can be found in TC++PL1 and D&E. However, the problem can seriously baffle a novice and undermine trust in the language. Language solutions have been proposed by Lois Goldtwaith, Francis Glassborow, Lawrence Crowl, and I [30]; some version may make it into C++0x.

I chose this example to illustrate the constraints imposed by compatibility. The problem could be eliminated by not providing default copy of objects of a class with pointer members; if people wanted to copy, they could supply a copy operator. However, that would break an unbelievable amount of code. In general, remedying long-standing problems is harder than it looks, especially if C compatibility enters into the picture. However, in this case, the existence of a destructor is a strong indicator that default copying would be wrong, so examples such as `String` could be reliably caught by the compiler.

8.5 Standard libraries

For the C++0x standard library, the stated aim was to make a much broader platform for systems programming. The June 2006 version “Standard Libraries Wish List” maintained by Matt Austern lists 68 suggested libraries including

- Container-based algorithms
- Random access to files
- Safe STL (completely range checked)
- File system access
- Linear algebra (vectors, matrices, etc.)
- Date and time

- Graphics library
- Data compression
- Unicode file names
- Infinite-precision integer arithmetic
- Uniform use of `std::string` in the library
- Threads
- Sockets
- Comprehensive support for unicode
- XML parser and generator library
- Graphical user interface
- Graph algorithms
- Web services (SOAP and XML bindings)
- Database support
- Lexical analysis and parsing

There has been work on many of these libraries, but most are postponed to post-C++0x library TRs. Sadly, this leaves many widely desired and widely used library components unstandardized. In addition, from observing people struggling with C++98, I also had high hopes that the committee would take pity on the many new C++ programmers and provide library facilities to support novices from a variety of backgrounds (not just beginning programmers and refugees from C). I have low expectations for the most frequently requested addition to the standard library: a standard GUI (Graphical User Interface; see §1 and §5.5).

The first new components of the C++0x standard library are those from the library TR (§6.2). All but one of the components were voted in at the spring 2006 meeting in Berlin. The special mathematical functions were considered to specialized for the vast majority of C++ programmers and were spun off to become a separate ISO standard.

In addition to the more visible work on adding new library components, much work in the library working group focuses on minor improvements to existing components, based on experience, and on improved specification of existing components. The accumulative effect of these minor improvements is significant.

The plan for 2007 includes going over the standard library to take advantage of new C++0x features. The first and most obvious example is to add rvalue initializers [57] (primarily the work of Howard Hinnant, Peter Dimov, and Dave Abrahams) to significantly improve the performance of heavily used components, such as `vector`, that occasionally have to move objects around. Assuming that concepts (§8.3.3) make it into C++0x, their use will revolutionize the specification of the STL part of the library (and other templated components). Similarly, the standard containers, such as `vector` should be augmented with sequence constructors to allow them to accept initializer lists (§8.3.1). Generalized constant expressions (`constexpr`, primarily the work

of Gabriel Dos Reis and I [39]) will allow us to define simple functions, such as the ones defining properties of types in the `numeric_limits` and the `bitset` operators, so that they are usable at compile time. The variadic template mechanism (§8.3.1) dramatically simplifies interfaces to standard components, such as `tuple`, that can take a wide variety of template arguments. This has significant implications on the usability of these standard library components in performance-critical areas.

Beyond that, only a threads library seems to have gathered enough support to become part of C++0x. Other components that are likely to become part of another library TR (TR2) are:

- File system library — platform-independent file system manipulation [32]
- Date and time library [45]
- Networking — sockets, TCP, UDP, multicast, iostreams over TCP, and more [80]
- `numeric_cast` — checked conversions [22]

The initial work leading up to the proposals and likely votes mentioned here has been the main effort of the library working group 2003-06. The networking library has been used commercially for applications on multiple continents for some time. At the outset, Matt Austern (initially AT&T, later Apple) was the chair; now Howard Hinnant (initially Metrowerks, later Apple) has that difficult job.

In addition, the C committee is adopting a steady stream of technical reports, which must be considered and (despite the official ISO policy that C and C++ are distinct languages) will probably have to be adopted into the C++ library even though they — being C-style — haven't benefited from support from C++ language features (such as user-defined types, overloading, and templates). Examples are decimal floating-point and unicode built-in types with associated operations and functions.

All in all, this is a massive amount of work for the couple of dozen volunteers in the library working group, but it is not quite the “ambitious and opportunistic” policy that I had hoped for in 2001 (§8). However, people who scream for more (such as me) should note that even what's listed above will roughly double the size of the standard library. The process of library TRs is a hope for the future.

8.6 Concurrency

Concurrency cannot be perfectly supported by a library alone. On the other hand, the committee still considers language-based approaches to concurrency, such as is found in Ada and Java, insufficiently flexible (§5.5). In particular, C++0x must support current operating-system thread library approaches, such as POSIX threads and Windows threads.

Consequently, the work on concurrency is done by an ad hoc working group straddling the library-language divide. The approach taken is to carefully specify a machine model

for C++ that takes into account modern hardware architectures [14] and to provide minimal language primitives:

- thread local storage [29]
- atomic types and operations [13]

The rest is left to a threads library. The current threads library draft is written by Howard Hinnant [58].

This “concurrency effort” is led by Hans Boehm (Hewlett-Packard) and Lawrence Crowl (formerly Sun, currently Google). Naturally, compiler writers and hardware vendors are most interested and supportive in this. For example, Clark Nelson from Intel redesigned the notion of sequencing that C++ inherited from C to better fit C++ on modern hardware [89]. The threads library will follow the traditions of Posix threads [19], Windows threads [163], and libraries based on those, such as `boost::thread` [16]. In particular, the C++0x threads library will not try to settle every issue with threading; instead it will provide a portable set of core facilities but leave some tricky issues system dependent. This is necessary to maintain compatibility both with earlier libraries and with the underlying operating systems. In addition to the typical thread library facilities (such as lock and mutex), the library will provide thread pools and a version of “futures” [56] as a higher-level and easier-to-use communications facility: A future can be described as a placeholder for a value to be computed by another thread; synchronization potentially happens when the future is read. Prototype implementations exist and are being evaluated [58].

Unfortunately, I had to accept that my hopes of direct support for distributed programming are beyond what we can do for C++0x.

9. Retrospective

This retrospective looks back with the hope of extracting lessons that might be useful in the future:

- Why did C++ succeed?
- How did the standards process serve the C++ community?
- What were the influences on C++ and what has been its impact?
- Is there a future for the ideals that underlie C++?

9.1 Why Did C++ Succeed?

That's a question thoughtful people ask me a few times every year. Less thoughtful people volunteer imaginative answers to it on the web and elsewhere most weeks. Thus, it is worth while to both briefly answer the question and to contradict the more popular myths. Longer versions of the answer can be found in D&E [121], in my first HOPL paper [120], and in sections of this paper (e.g., §1, §7, and §9.4). This section is basically a summary.

C++ succeeded because

- *Low-level access plus abstraction*: The original conception of C++, “C-like for systems programming plus Simula-like for abstraction,” is a most powerful and useful idea (§7.1, §9.4).
- *A useful tool*: To be useful a language has to be complete and fit into the work environments of its users. It is neither necessary nor sufficient to be the best in the world at one or two things (§7.1).
- *Timing*: C++ was not the first language to support object-oriented programming, but from day one it was available as a useful tool for people to try for real-world problems.
- *Non-proprietary*: AT&T did not try to monopolize C++. Early on, implementations were relatively cheap (e.g., \$750 for educational institutions) and in 1989 all rights to the language were transferred to the standard bodies (ANSI and later ISO). I and others from Bell Labs actively helped non-AT&T C++ implementers to get started.
- *Stable*: A high degree (but not 100%) of C compatibility was considered essential from day one. A higher degree of compatibility (but still not 100%) with earlier implementations and definitions was maintained throughout. The standards process was important here (§3.1, §6).
- *Evolving*: New ideas have been absorbed into C++ throughout (e.g., exceptions, templates, the STL). The world evolves, and so must a living language. C++ didn’t just follow fashion; occasionally, it led (e.g., generic programming and the STL). The standards process was important here (§4.1, §5, §8).

Personally, I particularly like that the C++ design never aimed as solving a clearly enumerated set of problems with a clearly enumerated set of specific solutions: I’d never design a tool that could only do what I wanted [116]. Developers tend not to appreciate this open-endedness and emphasis on abstraction until their needs change.

Despite frequent claims, the reason for C++’s success was *not*:

- *Just luck*: I am amazed at how often I hear that claimed. I fail to understand how people can imagine that I and others involved with C++ could — by pure luck — repeatedly have provided services preferred by millions of systems builders. Obviously an element of luck is needed in any endeavor, but to believe that mere bumbling luck can account for the use of C++ 1.0, C++ 2.0, ARM C++, and C++98 is stretching probabilities quite a bit. Staying lucky for 25 years can’t be easy. No, I didn’t imagine every use of templates or of the STL, but I aimed for generality (§4.1.2).
- *AT&T’s marketing might*: That’s funny! All other languages that have ever competed commercially with C++ were better financed. AT&T’s C++ marketing budget for 1985–1988 (inclusive) was \$5,000 (out of which we

only managed to spend \$3,000; see D&E). At the first OOPSLA, we couldn’t afford to bring a computer, we had no flyers, and not a single marketer (we used a chalkboard, a signup sheet for research papers, and researchers). In later years, things got worse.

- *It was first*: Well, Ada, Eiffel, Objective C, Smalltalk and various Lisp dialects were all available commercially before C++. For many programmers, C, Pascal, and Modula-2 were serious contenders.
- *Just C compatibility*: C compatibility helped — as it should because it was a design aim. However, C++ was never just C and it earned the hostility from many C devotees early on for that. To meet C++’s design aims, some incompatibilities were introduced. C later adopted several C++ features, but the time lag ensured that C++ got more criticism than praise for that (§7.6). C++ was never an “anointed successor to C”.
- *It was cheap*: In the early years, a C++ implementation was relatively cheap for academic institutions, but about as expensive as competitive languages for commercial use (e.g., about \$40,000 for a commercial source license). Later, vendors (notably Sun and Microsoft) tended to charge more for C++ than for their own proprietary languages and systems.

Obviously, the reasons for success are as complex and varied as the individuals and organizations that adopted C++.

9.2 Effects of the Standards Process

Looking back on the ’90s and the first years of the 21st century, what questions can we ask about C++ and the standards committee’s stewardship of the language?

- Is C++98 better than ARM C++?
- What was the biggest mistake?
- What did C++ get right?
- Can C++ be improved? and if so, how?
- Can the ISO standards process be improved? and if so, how?

Yes, C++98 is a better language (with a much better library) than ARM C++ for the kinds of problems that C++ was designed to address. Thus, the efforts of the committee must be deemed successful. In addition, the population of C++ programmers has grown by more than an order of magnitude (from about 150,000 in 1990 to over 3 million in 2004; §1) during the stewardship of the committee. Could the committee have done better? Undoubtedly, but exactly how is hard to say. The exact conditions at the time where decisions are made are hard to recall (to refresh your memory, see D&E [121]).

So what was the biggest mistake? For the early years of C++, the answer was easy (“not shipping a larger/better foundation library with the early AT&T releases” [120]).

However, for the 1991-1998 period, nothing really stands out — at least if we have to be realistic. If we don't have to be realistic, we can dream of great support for distributed systems programming, a major library including great GUI support (§5.5), standard platform ABIs, elimination of over-laborate design warts that exist only for backward compatibility, etc. To get a serious candidate for regret, I think we have to leave the technical domain of language features and library facilities and consider social factors. The C++ community has no real center. Despite the major and sustained efforts of its members, the ISO C++ committee is unknown or incredibly remote to huge numbers of C++ users. Other programming language communities have been better at maintaining conferences, websites, collaborative development forums, FAQs, distribution sites, journals, industry consortiums, academic venues, teaching establishments, accreditation bodies, etc. I have been involved in several attempts to foster a more effective C++ community, but I must conclude that neither I nor anyone else have been sufficiently successful in this. The centrifugal forces in the C++ world have been too strong: Each implementation, library, and tool supplier has their own effort that usually does a good job for their users, but also isolates those users. Conferences and journals have fallen victim to economic problems and to a curious (but pleasant) lack of dogmatism among the leading people in the C++ community.

What did C++ get right? First of all, the ISO standard was completed in reasonable time and was the result of a genuine consensus. That's the base of all current C++ use. The standard strengthened C++ as a multi-paradigm language with uncompromising support for systems programming, including embedded systems programming.

Secondly, generic programming was developed and effective ways of using the language facilities for generic programming emerged. This not only involved work on templates (e.g., concepts) but also required a better understanding of resource management (§5.3) and of generic programming techniques. Bringing generic programming to the mainstream will probably be the main long-term technical contribution of C++ in this time period.

Can C++ be improved? and if so, how? Obviously, C++ can be improved technically, and I also think that it can be improved in the context of its real-world use. The latter is much harder, of course, because that imposes Draconian compatibility constraints and cultural requirements. The C++0x effort is the major attempt in that direction (§8). In the longer term (C++1x?), I hope for perfect type safety and a general high-level model for concurrency.

The question about improving the standards process is hard to answer. On the one hand, the ISO process is slow, bureaucratic, democratic, consensus-driven, subject to objections from very small communities (sometimes a single person), lacking of focus (“vision”), and cannot respond rapidly to changes in the industry or academic fashions. On the other

hand, that process has been successful. I have sometimes summed up my position by paraphrasing Churchill: “the ISO standards process is the worst, except for all the rest”. What the standards committee seems to lack is a “secretariat” of a handful of technical people who could spend full time examining needs, do comparative studies of solutions, experiment with language and library features, and work out detailed formulation of proposals. Unfortunately, long-time service in such a “secretariat” could easily be career death for a first-rate academic or industrial researcher. On the other hand, such a secretariat staffed by second-raters would lead to massive disaster. Maybe a secretariat of technical people serving for a couple of years supporting a fundamentally democratic body such as the ISO C++ committee would be a solution. That would require stable funding, though, and standards organizations do not have spare cash.

Commercial vendors have accustomed users to massive “standard libraries”. The ISO standards process — at least as practiced by the C and C++ committees — has no way of meeting user expectations of a many hundred thousands of lines of free standard-library code. The Boost effort (§4.2) is an attempt to address this. However, standard libraries have to present a reasonably coherent view of computation and storage. Developers of such libraries also have to spend a huge amount of effort on important utility libraries of no significant intellectual interest: just providing “what everybody expects”. There are also a host of mundane specification issues that even the commercial suppliers usually skimp on — but a standard must address because it aims to support multiple implementations. A loosely connected group of volunteers (working nights) is ill equipped to deal with that. A multi-year mechanism for guidance is necessary for coherence (and for integrating novel ideas) as well as ways of getting “ordinary development work” done.

9.3 Influences and impact

We can look at C++ in many ways. One is to consider influences:

1. What were the major influences on C++?
2. What languages, systems, and techniques were influenced by C++?

Recognizing the influences on the C++ language features is relatively easy. Recognizing the influences on C++ programming techniques, libraries, and tools is far harder because there is so much more going on in those areas and the decisions are not channeled through the single point of the standards committee. Recognizing C++'s influences on other languages, libraries, tools, and techniques is almost impossible. This task is made harder by a tendency of intellectual and commercial rivalries to lead to a lack of emphasis on documenting sources and influences. The competition for market share and mind share does not follow the ideal rules of academic publishing. In particular, most major ideas

have multiple variants and sources, and people tend to refer to sources that are not seen as competitors — and for many new languages C++ is “the one to beat”.

9.3.1 Influences on C++

The major influences on early C++ were the programming languages C and Simula. Along the way further influences came from Algol68, BCPL, Clu, Ada, ML, and Modula-2 [121]. The language-technical borrowings tend to be accompanied by programming techniques related to those features. Many of the design ideas and programming techniques came from classical systems programming and from UNIX. Simula contributed not just language features, but ideas of programming technique relating to object-oriented programming and data abstraction. My emphasis on strong static type checking as a tool for design, early error detection, and run-time performance (in that order) reflected in the design of C++ came primarily from there. With generic programming and the STL, mainstream C++ programming received a solid dose of functional programming techniques and design ideas.

9.3.2 Impact of C++

C++’s main contribution was and is through the many systems built using it (§7 [137]). The primary purpose of a programming language is to help build good systems. C++ has become an essential ingredient in the infrastructure on which our civilization relies (e.g. telecommunications systems, personal computers, entertainment, medicine, and electronic commerce) and has been part of some of the most inspiring achievements of this time period (e.g., the Mars Rovers and the sequencing of the human genome).

C++ was preeminent in bringing object-oriented programming into the mainstream. To do so, the C++ community had to overcome two almost universal objections:

- Object-oriented programming is inherently inefficient
- Object-oriented programming is too complicated to be used by “ordinary programmers”

C++’s particular variant of OO was (deliberately) derived from Simula, rather than from Smalltalk or Lisp, and emphasized the role of static type checking and its associated design techniques. Less well recognized is that C++ also brought with it some non-OO data abstraction techniques and an emphasis on statically typed interfaces to non-OO facilities (e.g., proper type checking for plain old C functions; §7.6). When that was recognized, it was often criticized as “hybrid”, “transitory”, or “static”. I consider it a valuable and often necessary complement to the view of object-oriented programming focused on class hierarchies and dynamic typing.

C++ brought generic programming into the mainstream. This was a natural evolution of the early C++ emphasis on statically typed interfaces and brought with it a number of functional programming techniques “translated” from lists

and recursion to general sequences and iteration. Function templates plus function objects often take the role of higher-order functions. The STL was a crucial trendsetter. In addition, the use of templates and function objects led to an emphasis on static type safety in high-performance computing (§7.3) that hitherto had been missing in real-world applications.

C++’s influence on specific language features is most obvious in C, which has now “borrowed”:

- function prototypes
- const
- inline
- bool
- complex
- declarations as statements
- declarations in for-statement initializers
- // comments

Sadly, with the exception of the // comments (which I in turn borrowed from BCPL), these features were introduced into C in incompatible forms. So was void*, which was a joint effort between Larry Rosler, Steve Johnson, and me at Bell Labs in mid-1982 (see D&E [121]).

Generic programming, the STL, and templates have also been very influential on the design of other languages. In the 1980s and 1990s templates and the programming techniques associated with them were frequently scorned as “not object-oriented”, too complicated, and too expensive; workarounds were praised as “simple”. The comments had an interesting similarity to the early community comments on object-oriented programming and classes. However, by 2005, both Java and C# have acquired “generics” and statically typed containers. These generics look a lot like templates, but Java’s are primarily syntactic sugar for abstract classes and far more rigid than templates. Though still not as flexible or efficient as templates, the C# 2.0 generics are better integrated into the type system and even (as templates always did) offer a form of specialization.

C++’s success in real-world use also led to influences that are less beneficial. The frequent use of the ugly and irregular C/C++ style of syntax in modern languages is not something I’m proud of. It is, however, an excellent indicator of C++ influence — nobody would come up with such syntax from first principles.

There is obvious C++ influence in Java, C#, and various scripting languages. The influence on Ada95, COBOL, and Fortran is less obvious, but definite. For example, having an Ada version of the Booch components [15] that could compare in code size and performance with the C++ version was a commonly cited goal. C++ has even managed to influence the functional programming community (e.g. [59]). However, I consider those influences less significant.

One of the key areas of influence has been on systems for intercommunication and components, such as CORBA and COM, both of which have a fundamental model of interfaces derived from C++’s abstract classes.

9.4 Beyond C++

C++ will be the mainstay of much systems development for years to come. After C++0x, I expect to see C++1x as the language and its community adapts to new challenges. However, where does C++ fit philosophically among current languages; in other words, ignoring compatibility, what is the essence of C++? C++ is an approximation to ideals, but obviously not itself the ideal. What key properties, ideas, and ideals might become the seeds of new languages and programming techniques? C++ is

- a set of low-level language mechanisms (dealing directly with hardware)
- combined with powerful compositional abstraction mechanisms
- relying on a minimal run-time environment.

In this, it is close to unique and much of its strength comes from this combination of features. C is by far the most successful traditional systems programming language; the dominant survivor of a large class of popular and useful languages. C is close to the machine in exactly the way C++ is (§2, §6.1), but C doesn’t provide significant abstraction mechanisms. To contrast, most “modern languages” (such as Java, C#, Python, Ruby, Haskell, and ML) provide abstraction mechanisms, but deliberately put barriers between themselves and the machine. The most popular rely on a virtual machine plus a huge run-time support system. This is a great advantage when you are building an application that requires the services offered, but a major limitation on the range of application areas (§7.1). It also ensures that every application is a part of a huge system, making it hard to understand completely and impossible to make correct and well performing in isolation.

In contrast, C++ and its standard library can be implemented in itself (the few minor exceptions to this are the results of compatibility requirements and minor specification mistakes). C++ is complete both as a mechanism for dealing with hardware and for efficient abstraction from the close-to-hardware levels of programming. Occasionally, optimal use of a hardware feature requires a compiler intrinsic or an inline assembler insert, but that does not break the fundamental model; in fact, it can be seen as an integral part of that model. C++’s model of computation is that of hardware, rather than a mathematical or ad hoc abstraction.

Can such a model be applied to future hardware, future systems requirements, and future application requirements? I consider it reasonably obvious that in the absence of compatibility requirements, a new language on this model can be much smaller, simpler, more expressive, and more amenable

to tool use than C++, without loss of performance or restriction of application domains. How much smaller? Say 10% of the size of C++ in definition and similar in front-end compiler size. In the “Retrospective” chapter of D&E [121], I expressed that idea as “Inside C++, there is a much smaller and cleaner language struggling to get out”. Most of the simplification would come from generalization — eliminating the mess of special cases that makes C++ so hard to handle — rather than restriction or moving work from compile time to run time. But could a machine-near plus zero-overhead abstraction language meet “modern requirements”? In particular, it must be

- completely type safe
- capable of effectively using concurrent hardware

This is not the place to give a technical argument, but I’m confident that it could be done. Such a language would require a realistic (relative to real hardware) machine model (including a memory model) and a simple object model. The object model would be similar to C++’s: direct mapping of basic types to machine objects and simple composition as the basis for abstraction. This implies a form of pointers and arrays to support a “sequence of objects” basic model (similar to what the STL offers). Getting that type safe — that is, eliminating the possibility of invalid pointer use, range errors, etc. — with a minimum of run-time checks is a challenge, but there is plenty of research in software, hardware, and static analysis to give cause for optimism. The model would include true local variables (for user-defined types as well as built-in ones), implying support for “resource acquisition is initialization”-style resource management. It would not be a “garbage collection for everything” model, even though there undoubtedly would be a place for garbage collection in most such languages.

What kind of programming would this sort of language support? What kind of programming would it support better than obvious and popular alternatives? This kind of language would be a systems programming language suitable for hard real-time applications, device drivers, embedded devices, etc. I think the ideal systems programming language belongs to this general model — you need machine-near features, predictable performance, (type) safety, and powerful abstraction features. Secondly, this kind of language would be ideal for all kinds of resource-constrained applications and applications with large parts that fitted these two criteria. This is a huge and growing class of applications. Obviously, this argument is a variant of the analysis of C++’s strengths from §7.1.

What would be different from current C++? Size, type safety, and integral support for concurrency would be the most obvious differences. More importantly, such a language would be more amenable to reasoning and to tool use than C++ and languages relying on lots of run-time support. Beyond that, there is ample scope for improvement over C++

in both design details and specification techniques. For example, the integration between core language and standard library features can be much smoother; the C++ ideal of identical support for built-in and user-defined types could be completely achieved so that a user couldn't (without examining the implementation) tell which was which. It is also obvious that essentially all of such a language could be defined relative to an abstract machine (thus eliminating all incidental "implementation defined" behavior) without compromising the close-to-machine ideal: machine architectures have huge similarities in operations and memory models that can be exploited.

The real-world challenge to be met is to specify and implement correct systems (often under resource constraints and often in the presence of the possibility of hardware failure). Our civilization critically depends on software and the amount and complexity of that software is steadily increasing. So far, we (the builders of large systems) have mostly addressed that challenge by patch upon patch and incredible numbers of run-time tests. I don't see that approach continuing to scale. For starters, apart from concurrent execution, our computers are not getting faster to compensate for software bloat as they did for the last decades. The way to deal that I'm most optimistic about is a more principled approach to software, relying on more mathematical reasoning, more declarative properties, and more static verification of program properties. The STL is an example of a move in that direction (constrained by the limitations of C++). Functional languages are examples of moves in that direction (constrained by a fundamental model of memory that differs from the hardware's and underutilization of static properties). A more significant move in that direction requires a language that supports specification of desired properties and reasoning about them (concepts are a step in that direction). For this kind of reasoning, run-time resolution, run-time tests, and multiple layers of run-time mapping from code to hardware are at best wasteful and at worst serious obstacles. The ultimate strength of a machine-near, type-safe, and compositional abstraction model is that it provides the fewest obstacles to reasoning.

10. Acknowledgments

Obviously, I owe the greatest debt to the members of the C++ standards committee who worked — and still work — on the language and library features described here. Similarly, I thank the compiler and tools builders who make C++ a viable tool for an incredible range of real-world problems.

Thanks to Matt Austern, Paul A. Bistow, Steve Clamage, Greg Colvin, Gabriel Dos Reis, S. G. Ganesh, Lois Goldthwaite, Kevlin Henney, Howard Hinnant, Jaakko Järvi, Andy Koenig, Bronek Kozicki, Paul McJones, Scott Meyers, W. M. (Mike) Miller, Sean Parent, Tom Plum, Premanan M. Rao, Jonathan Schilling, Alexander Stepanov, Nicholas

Stroustrup, James Widman, and J. C. van Winkel for making constructive comments on drafts of this paper.

Also thanks to the HOPL-III reviewers: Julia Lawall, Mike Mahoney, Barbara Ryder, Herb Sutter, and Ben Zorn who — among other things — caused a significant increase in the introductory material, thus hopefully making this paper more self-contained and more accessible to people who are not C++ experts.

I am very grateful to Brian Kernighan and Doug McIlroy for all the time they spent listening to and educating me on a wide variety of language-related topics during the 1990s. A special thanks to Al Aho of Columbia University for lending me an office in which to write the first draft of this paper.

References

- [1] David Abrahams: *Exception-Safety in Generic Components*. M. Jazayeri, R. Loos, D. Musser (eds.): Generic Programming, Proc. of a Dagstuhl Seminar. Lecture Notes on Computer Science. Volume 1766. 2000. ISBN: 3-540-41090-2.
- [2] David Abrahams and Aleksey Gurtovoy: *C++ Template Meta-programming* Addison-Wesley. 2005. ISBN 0-321-22725-5.
- [3] Andrei Alexandrescu: *Modern C++ Design*. Addison-Wesley. 2002. ISBN 0-201-70431.
- [4] Ping An, Alin Jula, Silvius Rus, Steven Saunders, Tim Smith, Gabriel Tanase, Nathan Thomas, Nancy Amato, Lawrence Rauchwerger: *STAPL: An Adaptive, Generic Parallel C++ Library*. In Wkshp. on Lang. and Comp. for Par. Comp. (LCP), pp. 193-208, Cumberland Falls, Kentucky, Aug 2001.
- [5] AT&T C++ translator release notes. *Tools and Reusable Components*. 1989.
- [6] M. Austern: *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*. Addison Wesley. 1998. ISBN: 0-201-30956-4.
- [7] John Backus: *Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs*. Communications of the ACM 21, 8 (Aug. 1978).
- [8] J. Barreiro, R. Fraley, and D. Musser: *Hash Tables for the Standard Template Library*. Rensselaer Polytechnic Institute and Hewlett Packard Laboratories. February, 1995. <ftp://ftp.cs.rpi.edu/pub/stl/hashdoc.ps.Z>
- [9] Graham Birtwistle et al.: *SIMULA BEGIN*. Studentlitteratur. Lund. Sweden. 1979. ISBN 91-44-06212-5.
- [10] Jasmin Blanchette and Mark Summerfield: *C++ GUI Programming with Qt3*. Prentice Hall. 2004. ISBN 0-13-124072-2.
- [11] Hans-J. Boehm: *Space Efficient Conservative Garbage Collection*. Proc. ACM SIGPLAN '93 Conference on Programming Language Design and Implementation. ACM SIGPLAN Notices. June 1993. http://www.hpl.hp.com/personal/Hans_Boehm/gc/.
- [12] Hans-J. Boehm and Michael Spertus: *Transparent Garbage Collection for C++*. ISO SC22 WG21 TR NN1943==06-

- [13] Hans-J. Boehm: *An Atomic Operations Library for C++*. ISO SC22 WG21 TR N2047==06-0117.
- [14] Hans-J. Boehm: *A Less Formal Explanation of the Proposed C++ Concurrency Memory Model*. ISO SC22 WG21 TR N2138==06-0208.
- [15] Grady Booch: *Software Components with Ada*. Benjamin Cummings. 1988. ISBN 0-8053-0610-2.
- [16] The Boost collection of libraries. <http://www.boost.org>.
- [17] Walter Bright: *D Programming Language*. <http://www.digitalmars.com/d/>.
- [18] Peter A. Buhr and Glen Ditchfield: *Adding Concurrency to a Programming Language*. Proc. USENIX C++ Conference. Portland, OR. August 1992.
- [19] David Butenhof: *Programming With Posix Threads*. ISBN 0-201-63392-2. 1997.
- [20] T. Cargill: *Exception handling: A False Sense of Security*. The C++ Report, Volume 6, Number 9, November-December 1994.
- [21] D. Caromel et al.: *C++//*. In *Parallel programming in C++*. (Wilson and Lu, editors). MIT Press. 1996. ISBN 0-262-73118-5.
- [22] Fernando Cacciola: *A proposal to add a general purpose ranged-checked numeric_cast<>* (Revision 1). ISO SC22 WG21 TR N1879==05-0139.
- [23] CGAL: Computational Geometry Algorithm Library. <http://www.cgal.org/>.
- [24] Siva Challa and Artur Laksberg: *Essential Guide to Managed Extensions for C++*. Apress. 2002. ISBN: 1893115283.
- [25] K. M. Chandy and C. Kesselman: *Compositional C++: Compositional Parallel Programming*. Technical Report. California Institute of Technology. [Caltech CSTR: 1992.cs-tr-92-13].
- [26] Shigeru Chiba: *A Metaobject Protocol for C++*. Proc. OOPSLA'95. <http://www.csg.is.titech.ac.jp/~chiba/openc++.html>.
- [27] Steve Clamage and David Vandevoorde. Personal communications. 2005.
- [28] J. Coplien: *Multi-paradigm Design for C++*. Addison Wesley. 1998. ISBN 0-201-82467-1.
- [29] Lawrence Crowl: *Thread-Local Storage*. ISO SC22 WG21 TR N1966==06-0036.
- [30] Lawrence Crowl: *Defaulted and Deleted Functions*. ISO SC22 WG21 TR N2210==07-0070.
- [31] Krzysztof Czarnecki and Ulrich W. Eisenecker: *Generative Programming — Methods, Tools, and Applications*. Addison-Wesley, June 2000. ISBN 0-201-30977-7.
- [32] Beman Dawes: *Filesystem Library Proposal for TR2 (Revision 2)*. ISO SC22 WG21 TR N1934==06-0004.
- [33] Jeffrey Dean and Sanjay Ghemawat: *MapReduce: Simplified Data Processing on Large Clusters*. OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December, 2004.
- [34] D. Detlefs: *Garbage collection and run-time typing as a C++ library*. Proc. USENIX C++ conference. 1992.
- [35] M. Ellis and B. Stroustrup: *The Annotated C++ Reference Manual* ("The ARM") Addison Wesley. 1989.
- [36] Dinkumware: *Dinkum Abridged Library* <http://www.dinkumware.com/libdal.html>.
- [37] Gabriel Dos Reis and Bjarne Stroustrup: *Formalizing C++*. TAMU CS TR. 2005.
- [38] Gabriel Dos Reis and Bjarne Stroustrup: *Specifying C++ Concepts*. Proc. ACM POPL 2006.
- [39] Gabriel Dos Reis and Bjarne Stroustrup: *Generalized Constant Expressions* (Revision 3). ISO SC22 WG21 TR N1980==06-0050.
- [40] The Embedded C++ Technical Committee: *The Language Specification & Libraries Version*. WP-AM-003. Oct 1999 (<http://www.caravan.net/ec2plus/>).
- [41] Ecma International: *ECMA-372 Standard: C++/CLI Language Specification*. <http://www.ecma-international.org/publications/standards/Ecma-372.htm>. December 2005.
- [42] Boris Fomitch: *The STLport Story*. <http://www.stlport.org/doc/story.html>.
- [43] Eric Gamma, et al.: *Design Patterns*. Addison-Wesley. 1994. ISBN 0-201-63361-2.
- [44] R. Garcia, et al.: *A comparative study of language support for generic programming*. ACM OOPSLA 2003.
- [45] Jeff Garland: *Proposal to Add Date-Time to the C++ Standard Library*. ISO SC22 WG21 TR N1900==05-0160.
- [46] N.H. Gehani and W.D. Roome: *Concurrent C++: Concurrent Programming with Class(es)*. Software—Practice and Experience, 18(12):1157—1177, December 1988.
- [47] Geodesic: Great circle. Now offered by Veritas.
- [48] M. Gibbs and B. Stroustrup: *Fast dynamic casting*. Software—Practice&Experience. 2005.
- [49] William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir: *MPI: The Complete Reference — 2nd Edition: Volume 2 — The MPI-2 Extensions*. The MIT Press. 1998.
- [50] Keith E. Gorlen: *An Object-Oriented Class Library for C++ Programs*. Proc. USENIX C++ Conference. Santa Fe, NM. November 1987.
- [51] J. Gosling and H. McGilton: *The Java(tm) Language Environment: A White Paper*. <http://java.sun.com/docs/white/langenv/>.
- [52] Douglas Gregor, Jaakko Järvi, Jeremy G. Siek, Gabriel Dos Reis, Bjarne Stroustrup, and Andrew Lumsdaine: *Concepts: Linguistic Support for Generic Programming*. Proc. of ACM OOPSLA'06. October 2006.
- [53] D. Gregor and B. Stroustrup: *Concepts*. ISO SC22 WG21 TR N2042==06-0012.
- [54] D. Gregor, J. Järvi, G. Powell: *Variadic Templates* (Revision

- 3). ISO SC22 WG21 TR N2080==06-0150.
- [55] Douglas Gregor and Jaakko Järvi: *Variadic Templates for C++*. Proc. 2007 ACM Symposium on Applied Computing. March 2007.
- [56] R. H. Halstead: *MultiLisp: A Language for Concurrent Symbolic Computation*. TOPLAS. October 1985.
- [57] H. Hinnant, D. Abrahams, and P. Dimov: *A Proposal to Add an Rvalue Reference to the C++ Language*. ISO SC22 WG21 TR N1690==04-0130.
- [58] Howard E. Hinnant: *Multithreading API for C++0X - A Layered Approach*. ISO SC22 WG21 TR N2094==06-0164.
- [59] J. Hughes and J. Sparud: *Haskell++: An object-oriented extension of Haskell*. Proc. Haskell Workshop, 1995.
- [60] IA64 C++ ABI. <http://www.codesourcery.com/cxx-abi>.
- [61] IDC report on programming language use. 2004. <http://www.idc.com>.
- [62] *Standard for the C Programming Language*. ISO/IEC 9899. (“C89”).
- [63] *Standard for the C++ Programming Language*. ISO/IEC 14882. (“C++98”).
- [64] *Standard for the C Programming Language*. ISO/IEC 9899:1999. (“C99”).
- [65] International Organization for Standards: *The C Programming Language*. ISO/IEC 9899:2002. Wiley 2003. ISBN 0-470-84573-2.
- [66] International Organization for Standards: *The C++ Programming Language* ISO/IEC 14882:2003. Wiley 2003. ISBN 0-470-84674-7.
- [67] *Technical Report on C++ Performance*. ISO/IEC PDTR 18015.
- [68] *Technical Report on C++ Standard Library Extensions*. ISO/IEC PDTR 19768.
- [69] ISO SC22/WG21 website: <http://www.open-std.org/jtc1/sc22/wg21/>.
- [70] Sorin Istrail and 35 others: *Whole-genome shotgun assembly and comparison of human genome assemblies*. Proc. National Academy of Sciences. February, 2004. <http://www.pantherdb.org/>.
- [71] Jaakko Järvi: *Proposal for adding tuple type into the standard library*. ISO SC22 WG21 TR N1382==02-0040.
- [72] Jaakko Järvi, Gary Powell, and Andrew Lumsdaine: *The Lambda Library: Unnamed Functions in C++*. Software-Practice and Experience, 33:259-291, 2003.
- [73] J. Järvi, B. Stroustrup and G. Dos Reis: *Deducing the type of a variable from its initializer expression*. ISO SC22 WG21 TR N1894, Oct. 2005.
- [74] Jaakko Järvi, Douglas Gregor, Jeremiah Willcock, Andrew Lumsdaine, and Jeremy Siek: *Algorithm specialization in generic programming: challenges of constrained generics in C++*. Proc. PLDI 2006.
- [75] L. V. Kale and S. Krishnan: *CHARM++ in Parallel programming in C++*. (Wilson and Lu, editors). MIT Press. 1996. ISBN 0-262-73118-5.
- [76] Brian Kernighan and Dennis Richie: *The C Programming Language (“K&R”)*. Prentice Hall. 1978. ISBN 0-13-110163-3.
- [77] Brian Kernighan and Dennis Richie: *The C Programming Language (2nd edition) (“K&R2” or just “K&R”)*. Prentice Hall. 1988. ISBN 0-13-110362-8.
- [78] Brian Kernighan: *Why Pascal isn’t my favorite programming language*. AT&T Bell Labs technical report No. 100. July 1981.
- [79] Andrew Koenig and Bjarne Stroustrup: *Exception Handling for C++(revised)*. Proc. USENIX C++ Conference. San Francisco, CA. April 1990. Also, *Journal of Object-Oriented Programming*. July 1990.
- [80] Christopher Kohlhoff: *Networking Library Proposal for TR2*. ISO SC22 WG21 TR N2054==06-0124.
- [81] Mark A. Linton and Paul R. Calder: *The Design and Implementation of InterViews*. Proc. USENIX C++ Conference. Santa Fe, NM. November 1987.
- [82] A. Mishra et al.: *R++: Using Rules in Object-Oriented Designs*. Proc. OOPSLA-96. <http://www.research.att.com/sw/tools/r++/>.
- [83] W. G. O’Farrell et al.: *ABC++ in Parallel Programming in C++*. (Wilson and Lu, editors). MIT Press. 1996. ISBN 0-262-73118-5.
- [84] Thorsten Ottosen: *Proposal for new for-loop*. ISO SC22 WG21 TR N1796==05-0056.
- [85] Sean Parent: personal communications. 2006.
- [86] J. V. W. Reynolds et al.: *POOMA in Parallel Programming in C++*. (Wilson and Lu, editors). MIT Press. 1996. ISBN 0-262-73118-5.
- [87] Mike Mintz and Robert Ekendahl: *Hardware Verification with C++ — A Practitioner’s Handbook*. Springer Verlag. 2006. ISBN 0-387-25543-5.
- [88] Nathan C. Myers: *Traits: a new and useful template technique*. The C++ Report, June 1995.
- [89] C. Nelson and H.-J. Boehm: *Sequencing and the concurrency memory model*. ISO SC22 WG21 TR N2052==06-0122.
- [90] Mac OS X 10.1 November 2001 Developer Tools CD Release Notes: *Objective-C++*. <http://developer.apple.com/releasenotes/Cocoa/Objective-C++.html>
- [91] Leonie V. Rose and Bjarne Stroustrup: *Complex Arithmetic in C++*. Internal AT&T Bell Labs Technical Memorandum. January 1984. Reprinted in AT&T C++ Translator Release Notes. November 1985.
- [92] P. Rovner, R. Levin, and J. Wick: *On extending Modula-2 for building large integrated systems*. DEC research report #3. 1985.
- [93] J. Schilling: *Optimizing Away C++ Exception Handling*. ACM SIGPLAN Notices. August 1998.
- [94] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker,

- and Jack Dongarra: *MPI: The Complete Reference* The MIT Press. 1995. <http://www-unix.mcs.anl.gov/mpi/>.
- [95] D. C. Schmidt and S. D. Huston: *Network programming using C++*. Addison-Wesley Vol 1, 2001. Vol. 2, 2003. ISBN 0-201-60464-7 and ISBN 0-201-79525-6.
- [96] Jeremy G. Siek and Andrew Lumsdaine: *The Matrix Template Library: A Generic Programming Approach to High Performance Numerical Linear Algebra* ISCOPE '98. <http://www.osl.iu.edu/research/mtl>.
- [97] J. Siek et al.: *Concepts for C++*. ISO SC22 WG21 WG21-N1758.
- [98] Jeremy G. Siek and Walid Taha: *A Semantic Analysis of C++ Templates*. Proc. ECOOP 2006.
- [99] Yannis Smargakis: *Functional programming with the FC++ library*. ICFP'00.
- [100] Olaf Spinczyk, Daniel Lohmann, and Matthias Urban: *AspectC++: an AOP Extension for C++*. Software Developer's Journal. June 2005. <http://www.aspectc.org/>.
- [101] A. A. Stepanov and D. R. Musser: *The Ada Generic Library: Linear List Processing Packages*. Compass Series, Springer-Verlag, 1989.
- [102] A. A. Stepanov: *Abstraction Penalty Benchmark, version 1.2 (KAI)*. SGI 1992. Reprinted as Appendix D.3 of [67].
- [103] A. Stepanov and M. Lee: *The Standard Template Library*. HP Labs TR HPL-94-34. August 1994.
- [104] Alex Stepanov: Foreword to Siek, et al.: *The Boost Graph Library*. Addison-Wesley 2002. ISBN 0-21-72914-8.
- [105] Alex Stepanov: personal communications. 2004.
- [106] Alex Stepanov: personal communications. 2006.
- [107] Christopher Strachey: *Fundamental Concepts in Programming Languages*. Lecture notes for the International Summer School in Computer Programming, Copenhagen, August 1967.
- [108] Bjarne Stroustrup: *Classes: An Abstract Data Type Facility for the C Language*. Bell Laboratories Computer Science Technical Report CSTR-84. April 1980. Revised, August 1981. Revised yet again and published as [109].
- [109] Bjarne Stroustrup: *Classes: An Abstract Data Type Facility for the C Language*. ACM SIGPLAN Notices. January 1982. Revised version of [108].
- [110] B. Stroustrup: *Data abstraction in C*. Bell Labs Technical Journal. Vol 63. No 8 (Part 2), pp 1701-1732. October 1984.
- [111] B. Stroustrup: *A C++ Tutorial*. Proc. 1984 National Communications Forum. September, 1984.
- [112] B. Stroustrup: *The C++ Programming Language* ("TC++PL"). Addison-Wesley Longman. Reading, Mass., USA. 1986. ISBN 0-201-12078-X.
- [113] Bjarne Stroustrup: *What is Object-Oriented Programming?* Proc. 14th ASU Conference. August 1986. Revised version in Proc. ECOOP'87, May 1987, Springer Verlag Lecture Notes in Computer Science Vol 276. Revised version in *IEEE Software Magazine*. May 1988.
- [114] Bjarne Stroustrup: *An overview of C++*. ACM Sigplan Notices, Special Issue. October, 1986
- [115] B. Stroustrup and J. E. Shopiro: *A Set of Classes for Coroutine Style Programming*. Proc. USENIX C++ Workshop. November, 1987.
- [116] Bjarne Stroustrup: quote from 1988 talk.
- [117] Bjarne Stroustrup: *Parameterized Types for C++*. Proc. USENIX C++ Conference, Denver, CO. October 1988. Also, USENIX Computer Systems, Vol 2 No 1. Winter 1989.
- [118] B. Stroustrup: *The C++ Programming Language*, 2nd Edition ("TC++PL2" or just "TC++PL"). Addison-Wesley Longman. Reading, Mass., USA. 1991. ISBN 0-201-53992-6.
- [119] Bjarne Stroustrup and Dmitri Lenkov: *Run-Time Type Identification for C++*. The C++ Report. March 1992. Revised version. Proc. USENIX C++ Conference. Portland, OR. August 1992.
- [120] B. Stroustrup: *A History of C++: 1979-1991*. Proc ACM HOPL-II. March 1993. Also in Begin and Gibson (editors): *History of Programming Languages*. Addison-Wesley. 1996. ISBN 0-201-89502-1.
- [121] B. Stroustrup: *The Design and Evolution of C++*. ("D&E"). Addison-Wesley Longman. Reading Mass. USA. 1994. ISBN 0-201-54330-3.
- [122] B. Stroustrup: *Why C++ is not just an object-oriented programming language*. Proc. ACM OOPSLA 1995.
- [123] B. Stroustrup: *Proposal to Acknowledge that Garbage Collection for C++ is Possible*. WG21/N0932 X3J16/96-0114.
- [124] B. Stroustrup: *The C++ Programming Language*, 3rd Edition ("TC++PL3" or just "TC++PL"). Addison-Wesley Longman. Reading, Mass., USA. 1997. ISBN 0-201-88954-4.
- [125] B. Stroustrup: *Learning Standard C++ as a New Language*. The C/C++ Users Journal. May 1999.
- [126] B. Stroustrup: *The C++ Programming Language*, Special Edition ("TC++PL"). Addison-Wesley, February 2000. ISBN 0-201-70073-5.
- [127] B. Stroustrup: *C and C++: Siblings, C and C++: A Case for Compatibility, C and C++: Case Studies in Compatibility*. The C/C++ Users Journal. July, August, and September 2002.
- [128] B. Stroustrup: *Why can't I define constraints for my template parameters?*. http://www.research.att.com/~bs/bs_faq2.html#constraints.
- [129] B. Stroustrup and G. Dos Reis: *Concepts — Design choices for template argument checking*. ISO SC22 WG21 TR N1522. 2003.
- [130] B. Stroustrup: *Concept checking — A more abstract complement to type checking*. ISO SC22 WG21 TR N1510. 2003.
- [131] B. Stroustrup: *Abstraction and the C++ machine model*. Proc. ICES'04. December 2004.
- [132] B. Stroustrup, G. Dos Reis: *A concept design* (Rev. 1). ISO

- [133] B. Stroustrup: *A rationale for semantically enhanced library languages*. ACM LCSD05. October 2005.
- [134] B. Stroustrup and G. Dos Reis: *Supporting SELL for High-Performance Computing*. LCPC05. October 2005.
- [135] Bjarne Stroustrup and Gabriel Dos Reis: *Initializer lists*. ISO SC22 WG21 TR N1919=05-0179.
- [136] B. Stroustrup: *C++ pages*. <http://www.research.att.com/~bs/C++.html>.
- [137] B. Stroustrup: *C++ applications*. <http://www.research.att.com/~bs/applications.html>.
- [138] H. Sutter, D. Miller, and B. Stroustrup: *Strongly Typed Enums* (revision 2). ISO SC22 WG21 TR N2213==07-0073.
- [139] H. Sutter and B. Stroustrup: *A name for the null pointer: nullptr* (revision 2). ISO SC22 WG21 TR N1601==04-0041.
- [140] Herb Sutter: *A Design Rationale for C++/CLI*, Version 1.1 — February 24, 2006 (later updated with minor editorial fixes). <http://www.gotw.ca/publications/C+CLIRationale.pdf>.
- [141] Numbers supplied by Trolltech. January 2006.
- [142] UK C++ panel: *Objection to Fast-track Ballot ECMA-372 in JTC1 N8037*. <http://public.research.att.com/~bs/uk-objections.pdf>. January 2006.
- [143] E. Unruh: *Prime number computation*. ISO SC22 WG21 TR N462==94-0075.
- [144] D. Vandevoride and N. M. Josuttis: *C++ Templates — The Complete Guide*. Addison-Wesley. 2003. ISBN 0-201-73884-2.
- [145] D. Vandevoride: *Right Angle Brackets* (Revision 1). ISO SC22 WG21 TR N1757==05-0017 .
- [146] D. Vandevoride: *Modules in C++* (Version 3). ISO SC22 WG21 TR N1964==06-0034.
- [147] Todd Veldhuizen: *expression templates*. C++ Report Magazine, Vol 7 No. 4, May 1995.
- [148] Todd Veldhuizen: *Template metaprogramming*. The C++ Report, Vol. 7 No. 5. June 1995.
- [149] Todd Veldhuizen: *Arrays in Blitz++*. Proceedings of the 2nd International Scientific Computing in Object Oriented Parallel Environments (ISCOPE'98). 1998.
- [150] Todd Veldhuizen: *C++ Templates are Turing Complete* 2003.
- [151] Todd L. Veldhuizen: *Guaranteed Optimization for Domain-Specific Programming*. Dagstuhl Seminar of Domain-Specific Program Generation. 2003.
- [152] Andre Weinand et al.: *ET++ — An Object-Oriented Application Framework in C++*. Proc. OOPSLA'88. September 1988.
- [153] Gregory V. Wilson and Paul Lu: *Parallel programming using C++*. Addison-Wesley. 1996.
- [154] P.M. Woodward and S.G. Bond: *Algol 68-R Users Guide*. Her Majesty's Stationery Office, London. 1974. ISBN 0-11-771600-6.
- [155] *The Visualization Toolkit, An Object-Oriented Approach To 3D Graphics, 3rd edition*, ISBN 1-930934-12-2. <http://public.kitware.com/VTK>.
- [156] *FLTK: Fast Light Toolkit*. <http://www.fltk.org/>.
- [157] Lockheed Martin Corporation: *Joint Strike Fighter air vehicle coding standards for the system development and demonstration program*. Document Number 2RDU00001 Rev C. December 2005. <http://www.research.att.com/~bs/JSF-AV-rules.pdf>.
- [158] Microsoft Foundation Classes.
- [159] *Smartwin++: An Open Source C++ GUI library*. <http://smartwin.sourceforge.net/>.
- [160] *WTL: Windows Template Library — A C++ library for developing Windows applications and UI components*. <http://wtl.sourceforge.net>.
- [161] *gtkmm: C++ Interfaces for GTK+ and GNOME*. <http://www.gtkmm.org/>
- [162] *wxWidgets: A cross platform GUI library*. <http://wxwidgets.org/>.
- [163] Windows threads: *Processes and Threads*. http://msdn.microsoft.com/library/en-us/dllproc/base/processes_and_threads.asp.
- [164] Wikipedia: *Java (Sun) — Early history*. [http://en.wikipedia.org/wiki/Java_\(Sun\)](http://en.wikipedia.org/wiki/Java_(Sun)).

Statecharts in the Making: A Personal Account

David Harel

The Weizmann Institute of Science
Rehovot, ISRAEL 76100
dharel@weizmann.ac.il

Abstract

This paper is a highly personal and subjective account of how the language of statecharts came into being. The main novelty of the language is in being a fully executable visual formalism intended for capturing the behavior of complex real-world systems, and an interesting aspect of its history is that it illustrates the advantages of theoreticians venturing out into the trenches of the real world, "dirtying their hands" and working closely with the system's engineers. The story is told in a way that puts statecharts into perspective and discusses the role of the language in the emergence of broader concepts, such as visual formalisms in general, reactive systems, model-driven development, model executability and code generation.

Some of these ideas are the general notion of a **visual formalism**, the identification of the class of **reactive systems** and the arguments for its significance and special character, the notion of **model-based development**, of which the UML is one of the best-known products, the concept of **model executability** and evidence of its feasibility, whereby high-level behavioral models (especially graphical ones) can and should be executed just like conventional computer programs, and the related concept of full **code generation**, whereby these high-level models are translated — actually, compiled down — into running code in a conventional language. The claim is not that none of these concepts was ever contemplated before statecharts, but rather that they became identified and pinpointed as part and parcel of the work on statecharts, and were given convincing support and evidence as a result thereof.

1. Introduction

The invitation to write a paper on statecharts for this conference on the history of programming languages produces mixed feelings of pleasant apprehension. Pleasant because being invited to write this paper means that statecharts are considered to be a programming language. They are executable, compilable and analyzable, just like programs in any "real" programming language, so that what we have here is not "merely" a specification language or a medium for requirements documentation. The apprehension stems from the fact that writing a historical paper about something you yourself were heavily involved in is hard; the result is bound to be very personal and idiosyncratic, and might sound presumptuous. In addition to accuracy, the paper must also try to be of interest to people other than its author and his friends...

The decision was to take the opportunity to put the language into a broader perspective and, in addition to telling its "story", to discuss some of the issues that arose around it. An implicit claim here is that whatever specific vices and virtues statecharts possess, their emergence served to identify and solidify a number of ideas that are of greater significance than one particular language.

Permission to make digital/hard copy of part of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date of appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Permission may be requested from the Publications Dept., ACM, Inc., 2 Penn Plaza, New York, NY 11201-0701, USA, fax:+1(212) 869-0481, permissions@acm.org

©2007 ACM 978-1-59593-766-7/2007/06-ART5 \$5.00

DOI 10.1145/1238844.1238849

<http://doi.acm.org/10.1145/1238844.1238849>

2. Pre-1982

I am not a programming languages person. In fact, the reader might be surprised to learn that the only programming languages I know reasonably well are PL/I and Basic.... I also enjoyed APL quite a bit at the time. However, even in classical languages like Fortran, PASCAL or C, not to mention more modern languages like C++ and Java, I haven't really done enough programming to be considered any kind of expert. Actually, nothing really qualifies me as a programming language researcher or developer. Prior to statecharts I had published in programming language venues, such as *POPL*, the *ACM Symposium on Principles of Programming Languages*, but the papers were about principles and theory, not about languages.... They mostly had to do with the logics of programs, their expressive power and axiomatics, and their relevance to correctness and verification.

In 1977, while at MIT working on my PhD, I had the opportunity to take a summer job at a small company in Cambridge, MA, called Higher Order Software (HOS), owned and run by Margaret Hamilton and Saydean Zeldin. They had a method for specifying software that took the form of trees of functions — a sort of functional decomposition if you will — that had to adhere to a set of six well-formedness axioms [HZ76]. We had several interesting discussions, sometimes arguments, one of which had to do with verification. When asked how they recommend that one verify the correctness of a system described using their method, the answers usually related to validating the appropriateness of the syntax. When it came to true verification, i.e., making sure that the system does what you expect it to, what they were saying in a nutshell was, "Oh, that's not a problem at all in our method because we don't allow programs that do not satisfy their

requirements." In other words, they were claiming to have "solved" the issue of verification by virtue of disallowing incorrect programs in the language. Of course, this only passes the buck: The burden of verifying correctness now lies with the syntax checker...

This attitude towards correctness probably had to do with the declarative nature of the HOS approach, whereas had they constructed their method as an executable language the verification issue could not have been sidetracked in this way and would have had to be squarely confronted. Still, the basic tree-like approach of the HOS method was quite appealing, and was almost visual in nature. As a result, I decided to see if the technical essence of this basic idea could be properly defined, hopefully resulting in a semantically sound, and executable, language for functions, based on an and/or functional decomposition. The "and" was intended as a common generalization of concurrency and sequentiality (you must do *all* these things) and the "or" a generalization of choice and branching (you must do *at least* one of these things). This was very similar to the then newly introduced notion of **alternation**, which had been added to classical models of computation in the theory community by Chandra, Kozen and Stockmeyer [CKS81] and about which I recall having gotten very excited at the time. Anyway, the result of this effort was a paper titled *And/Or Programs: A New Approach to Structured Programming*, presented in 1979 at an IEEE conference on reliable software (it later appeared in final form in ACM TOPLAS) [H79&80].

After the presentation at the conference, Dr. Jonah Lavi (Loeb) from the Israel Aircraft Industries (IAI) wanted to know if I was planning to return to Israel (at the time I was in the midst of a postdoctoral position — doing theory — at IBM's Yorktown Heights Research Center), and asked if I'd consider coming to work for the IAI. My response was to politely decline, since the intention was indeed to return within a year or so, but to academia to do research and teaching. This short conversation turned out to be crucial to the statechart story, as will become clear shortly.

3. December 1982 to mid 1983: The Avionics Motivation

We cut now to December 1982. At this point I had already been on the faculty of the Weizmann Institute of Science in Israel for two years. One day, the same Jonah Lavi called, asking if we could meet. In the meeting, he described briefly some severe problems that the engineers at IAI seemed to have, particularly mentioning the effort underway at IAI to build a home-made fighter aircraft, which was to be called the Lavi (no connection with Jonah's surname). The most difficult issues came up, he said, within the Lavi's avionics team. Jonah himself was a methodologist who did not work on a particular project; rather, he was responsible within the IAI for evaluating and bringing in software engineering tools and methods. He asked whether I would be willing to consult on a one-day-per-week basis, to see whether the problems they were having could be solved.

In retrospect, that visit turned out to be a real turning point for me. Moreover, luck played an important part too,

since I feel that Jonah Lavi had no particular reason to prefer me to any other computer scientist, except for the coincidence of his happening to have heard that lecture on and/or programs a few years earlier. Whatever the case, I agreed to do the consulting, having for a long time harbored a never-consummated dream, or "weakness", for piloting, especially fighter aircraft.

And so, starting in December 1982, for several months, Thursday became my consulting day at the IAI. The first few weeks of this were devoted to sitting down with Jonah, in his office, trying to understand from him what the issues were. After a few such weeks, having learnt a lot from Jonah, whose broad insights into systems and software were extremely illuminating, I figured it was time to become exposed to the real project and the specific difficulties there. In fact, at that point I hadn't yet met the project's engineers at all. An opportunity for doing so arrived, curiously enough, as a result of a health problem that prevented Jonah from being in the office for a few weeks, so that our thinking and talking had to be put on hold. The consulting days of that period were spent, accompanied by Jonah's able assistant Yitzhak Shai, working with a select group of experts from the Lavi avionics team, among whom were Akiva Kaspi and Yigal Livne.

These turned out to be an extremely fruitful few weeks, during which I was able to get a more detailed first-hand idea about the problem and to take the first steps in proposing a solution. We shall get to that shortly, but first some words about avionics.

An avionics system is a great example of what Amir Pnueli and I later identified as a **reactive system** [HP85]. The aspect that dominates such a system is its reactivity; its event-driven, control-driven, event-response nature, often including strict time constraints, and often exhibiting a great deal of parallelism. A typical reactive system is not particularly data intensive or calculation-intensive. So what is/was the problem with such systems? In a nutshell, it is the need to provide a clear yet precise description of what the system does, or should do. Specifying its **behavior** is the real issue.

Here is how the problem showed up in the Lavi. The avionics team had many amazingly talented experts. There were radar experts, flight control experts, electronic warfare experts, hardware experts, communication experts, software experts, etc. When the radar people were asked to talk about radar, they would provide the exact algorithm the radar used in order to measure the distance to the target. The flight control people would talk about the synchronization between the controls in the cockpit and the flaps on the wings. The communications people would talk about the formatting of information traveling through the MuxBus communication line that runs lengthwise along the aircraft. And on and on. Each group had their own idiosyncratic way of thinking about the system, their own way of talking, their own diagrams, and their own emphases.

Then I would ask them what seemed like very simple specific questions, such as: "What happens when this button on the stick is pressed?" In way of responding, they would take out a two-volume document, written in structured natural language, each volume containing

something like 900 or 1000 pages. In answer to the question above, they would open volume B on page 389, at clause 19.11.6.10, where it says that if you press this button, such and such a thing occurs. At which point (having learned a few of the system's buzzwords during this day-a-week consulting period) I would say: "Yes, but is that true even if there is an infra-red missile locked on a ground target?" To which they would respond: "Oh no, in volume A, on page 895, clause 6.12.3.7, it says that in such a case this other thing happens." This to-and-fro Q&A session often continued for a while, and by question number 5 or 6 they were often not sure of the answer and would call the customer for a response (in this case some part of the Israeli Air Force team working with the IAI on the aircraft's desired specification). By the time we got to question number 8 or 9 even those people often did not have an answer! And, by the way, one of Jonah Lavi's motivations for getting an outside consultant was the bothersome fact that some of the IAI's subcontractors refused to work from these enormous documents, claiming that they couldn't understand them and were in any case not certain that they were consistent or complete.

In my naïve eyes, this looked like a bizarre situation, because it was obvious that someone, eventually, would make a decision about what happens when you press a certain button under a certain set of circumstances. However, that person might very well turn out to be a low-level programmer whose task it was to write some code for some procedure, and who inadvertently was making decisions that influenced crucial behavior on a much higher level. Coming, as I did, from a clean-slate background in terms of avionics (which is a polite way of saying that I knew nothing about the subject matter...), this was shocking. It seemed extraordinary that this talented and professional team *did* have answers to questions such as "What algorithm is used by the radar to measure the distance to a target?", but in many cases did *not* have the answers to questions that seemed more basic, such as "What happens when you press this button on the stick under all possible circumstances?".

In retrospect, the two only real advantages I had over the avionics people were these: (i) having had no prior expertise or knowledge about this kind of system, which enabled me to approach it with a completely blank state of mind and think of it any which way; and (ii) having come from a slightly more mathematically rigorous background, making it somewhat more difficult for them to convince me that a two-volume, 2000 page document, written in structured natural language, was a complete, comprehensive and consistent specification of the system's behavior.

In order to make this second point a little more responsibly, let us take a look at an example taken from the specification of a certain chemical plant. It involves a tiny slice of behavior that I searched for tediously in this document (which was about 700 pages long). I found this particular piece of behavior mentioned in three different places in the document. The first is from an early part, on security, and appeared around page 10 of the document:

Section 2.7.6: Security

"If the system sends a signal hot then send a message to the operator."

Later on, in a section on temperatures, which was around page 150 of the document, it says:

Section 9.3.4: Temperatures

"If the system sends a signal hot and T >60°, then send a message to the operator."

The real gem was in the third quote, which occurred somewhere around page 650 of the document, towards the end, in a section devoted to summarizing some critical aspects of the system. There it says the following:

Summary of critical aspects

"When the temperature is maximum, the system should display a message on the screen, unless no operator is on the site except when T <60°."

Despite being educated as a logician, I've never really been able to figure out whether the third of these is equivalent to, or implies, any of the previous two... But that, of course, is not the point. The point is that these excerpts were obviously written by three different people for three different reasons, and that such large documents get handed over to programmers, some more experienced than others, to write the code. It is almost certain that the person writing the code for this critical aspect of the chemical plant will produce something that will turn out to be problematic in the best case — catastrophic in the worst. In addition, keep in mind that these excerpts were found by an extensive search through the entire document to try find where this little piece of behavior was actually mentioned. Imagine our programmer having to do that repeatedly for whatever parts of the system he/she is responsible for, and then to make sense of it all.

The specification documents that the Lavi avionics group had produced at the Israel Aircraft Industries were no better; if anything, they were longer and more complex, and hence worse, which leads to the question of how such an engineering team should specify behavior of such a system in a intuitively clear and mathematically rigorous fashion. These two characteristics, clarity and rigor, will take on special importance as our story unfolds.

4. 1983: Statecharts Emerging

Working with the avionics experts every Thursday for several weeks was a true eye-opener. At the time there was no issue of inventing a new programming language. The goal was to try to find, or to invent for these experts, a means for simply saying what they seemed to have had in their minds anyway. Despite the fact that the simple "what happens" questions get increasingly more complicated to answer, it became very clear that these engineers knew a tremendous amount about the intended behavior of the system. They understood it, and they had answers to many of the questions about behavior. Other questions they hadn't had the opportunity to think about properly because the information wasn't well organized in their documents, or even, for that matter, in their minds. The goal was to

find a way to help take the information that was present collectively in their heads and put it on paper, so to speak, in a fashion that was both well organized and accurate.

Accordingly, the work progressed in the following way. A lot of time was spent getting them to talk; I kept asking questions, prodding them to state clearly how the aircraft behaves under certain sets of circumstances. Example: "What are the radar's main activities in air-ground mode when the automatic pilot is on?" They would talk and we would have discussions, trying to make some coherent sense of the information that piled up.

I became convinced from the start that the notion of a state and a transition to a new state was fundamental to their thinking about the system. (This insight was consistent with some of the influential work David Parnas had been doing for a few years on the A-7 avionics [HKSP78].) They would repeatedly say things like, "When the aircraft is in air-ground mode and you press this button, it goes into air-air mode, but only if there is no radar locked on a ground target at the time". Of course, for anyone coming from computer science this is very familiar: what we really have here is a finite-state automaton, with its state transition table or state transition diagram. Still, it was pretty easy to see that just having one big state machine describing what is going on would be fruitless, and not only because of the number of states, which, of course, grows exponentially in the size of the system. Even more important seemed to be the pragmatic point of view, whereby a formalism in which you simply list all possible states and specify all the transitions between them is unstructured and non-intuitive; it has no means for modularity, hiding of information, clustering, and separation of concerns, and was not going to work for the kind of complex behavior in the avionics system. And if you tried to draw it visually you'd get spaghetti of the worst kind. It became obvious pretty quickly that it could be beneficial to come up with some kind of structured and hierarchical extension of the conventional state machine formalism.

So following an initial attempt at formalizing parts of the system using a sort of temporal logic-like notation (see Fig. 1)¹, I resorted to writing down the state-based behavior textually, in a kind of structured dialect made up on the fly that talked about states and their structure and the transitions between them. However, this dialect was hierarchical: inside a state there could be other states, and if you were in this state, and that event occurred, you would leave the current state and anything inside it and enter that other state, and so on. Fig. 2 shows an early example, from somewhere in early 1983, of one of these structured **state protocols**, or **statocols**, taken from my messy, scribbled IAI notebook.

As this was going on, things got increasingly complicated. The engineers would bring up additional pieces of the avionics behavior, and after figuring out how the new stuff related to the old, I would respond by extending the state-based structured description, often having to enrich the syntax in real time... When things got

a little more complicated, I would doodle on the side of the page to explain visually what was meant; some of this is visible on the right-hand side of Fig. 2. I clearly recall the first time I used visual encapsulation to illustrate to them the state hierarchy, and an arrow emanating from the higher level to show a compound "leave-any-state-inside" transition; see the doodling in Fig. 2 and the more orderly attempts in Fig. 3. And I also remember the first time I used side-by-side adjacency for orthogonal (concurrent) state components, denoted — after playing with two or three possible line styles — by a dashed line; see Fig. 4. However, it is important to realize that, at the time, these informal diagrams were drawn in order to explain what the nongraphical state protocols meant. The text was still the real thing and the diagrams were merely an aid.

After a few of these meetings with the avionics experts, it suddenly dawned on me that everyone around the table seemed to understand the back-of-napkin style diagrams a lot better and related to them far more naturally. The pictures were simply doing a much better job of setting down on paper the system's behavior, as understood by the engineers, and we found ourselves discussing the avionics and arguing about them over the diagrams, not the statocols. Still, the mathematician in me argued thus: "How could these doodled diagrams be better than the real mathematical-looking artifact?" (Observe Fig. 2 again, to see the two options side by side.) So it really took a leap of faith to be able to think: "Hmmm... couldn't the pictures be turned into the real thing, replacing, rather than supplementing, the textual structured programming-like formalism?" And so, over a period of a few weeks the scales tipped in favor of the diagrams. I gradually stopped using the text, or used it only to capture supplementary information inside the states or along transitions, and the diagrams became the actual specification we were constructing; see Figs. 5–9.

Of course, this had to be done in a responsible way, making sure that the emerging pictures were not just pictures; that they were not just doodling. They had to be rigorous, based on precise mathematical meaning. You couldn't just throw in features because they looked good and because the avionics team seemed to understand them. Unless the exact meaning of an intended feature was given, in any allowed context and under any allowed set of circumstances, it simply couldn't be considered.

This was how the basics of the language emerged. I chose to use the term **statecharts** for the resulting creatures, which was as of 1983 the only unused combination of "state" or "flow" with "chart" or "diagram".

5. On the Language Itself

Besides a host of other constructs, the two main ideas in statecharts are **hierarchy** and **orthogonality**, and these can be intermixed on all levels: You start out with classical finite-state machines (FSMs) and their state transition diagrams, and you extend them by a semantically meaningful hierarchical subsuming mechanism and by a notion of orthogonal simultaneity. Both of these are reflected in the graphics themselves, the hierarchy by encapsulation and the orthogonality by adjacent portions separated by a dashed line. Orthogonal components can

¹ Because of the special nature and size of some of the figures, I have placed them all at the end of the text, before the references.

cooperate and know about each other by several means, including direct sensing of the state status in another component or by actions. The cooperation mechanism — within a single statechart I should add — has a broadcasting flavor.

Transitions become far more elaborate and rich than in conventional FSMs. They can start or stop at any level of the hierarchy, can cross levels thereof, and in general can be hyperedges, since both sources and targets of transitions can contain sets of states. In fact, at any given point in time a statechart will be in a vector, or combination, of states, whose length is not fixed. Exiting and entering orthogonal components on the various levels of the hierarchy continuously changes the size of the state vector. Default states generalize start states, and they too can be level-crossing and of hyperedge nature. And the language has history connectors, conditions, selection connectors, and more. A transition can be labeled with an event and optionally also with a parenthesized condition, as well as with Mealy-like outputs, or actions. (Actions can also occur within states, in the Moore style.)

The fact that the technical part of the statecharts story started out with and/or programs is in fact very relevant. Encapsulated substates represent OR (actually this is XOR; exclusive or), and orthogonality is AND. Thus, a minimalist might view statecharts as a state-based language whose underlying structuring mechanism is simply that of classical alternation [CKS81]. Figs. 9 and 10 exemplify this connection by showing a state hierarchy for a part of the Lavi avionics statecharts and then the and/or tree I used to explain to the engineers in a different way what was actually going on.

In order to make this paper a little more technically informative, I will now carry out some self-plagiarism, stealing and then modifying some of the figures and explanations of the basic features of the language from the original statechart paper [H84&87]. However, the reader should not take the rest of this section as a tutorial on the language, or as a language manual. It is extremely informal, and extremely partial. I am also setting it in smaller font, and slightly indented, so that you can skip it completely if you want. For more complete accounts, please refer to [H84&87, HN89&96, HP91, HK04].

In way of introducing the state hierarchy, consider Fig. 11(i). It shows a very simple four-state chart. Notice, however, that event β takes the system to state B from either A or C, and also that δ takes the system to D from either of these. Thus, we can cluster A and C into a new **superstate**, E, and replace the two β transitions and the two δ ones by a single transition for each, as in Fig. 11(ii). The semantics of E is then the XOR of A and C; i.e., to be in state E one must be either in A or in C, but not in both. Thus E is really an abstraction of A and C, and its outgoing β and δ arrows capture two common properties of A and C; namely, that β leads from them to B and δ to D. The decision to have transitions that leave a superstate denote transitions leaving all substates turns out to be highly important, and is one of the main ways statecharts economize in the number of arrows.

Fig. 11 might also be approached from a different angle: first we might have decided upon the simple situation of Fig.

11(iii) and then state E could have been refined to consist of A and C, yielding Fig. 11(ii). Having decided to make this refinement, however, the transitions entering E in Fig. 11(iii), namely, α , δ and γ , become underspecified, as they do not say which of A or C is to be entered. This can be remedied in a number of ways. One is to simply extend them to point directly to A or C, as with the α -arrow entering A directly in Fig. 11(ii). Another is to use multi-level default entrances, as we now explain.

Fig. 11(i) has a start arrow pointing to state A. In finite automata this means simply that the automaton starts in state A. In statecharts this notion is generalized to that of a **default state**, which, in the context of Fig. 11 is taken to mean that as far as the 'outside' world is concerned A is the default state among A, B, C and D: if we are asked to enter one of these states but are not told which one to enter, the system is to enter A. In Fig. 11(i) this is captured in the obvious way, but in Fig. 11(ii) it is more subtle. The default arrow starts on the (topological) outside of the superstate E and enters A directly. This does not contradict the other default arrow in Fig. 11(ii), which is (topologically) wholly contained inside E and which leads to C. Its semantics is that if we somehow already entered E, but inside E we are not told where to go, the inner default is C, not A. This takes care of the two otherwise-underspecified transitions entering (and stopping at the borderline of) state E, those labeled δ and γ , emanating from B and D, respectively, and which indeed by Fig. 11(i) are to end up in C, not in A. Thus, Figs. 11(i) and 11(ii) are totally equivalent in their information, whereas Fig. 11(iii) contains less information and is thus an abstraction.

Besides the default entrance, there are other special ways to enter states, including **conditional** entries, specified by a circled C, and **history** entrances, specified by a circled H. The latter is particularly interesting, as it allows one to specify entrance to the substate most recently visited within the state in question, and thus caters for a (theoretically very limited, but in practice useful) kind of memory. In both of these, the connector's location within the state hierarchy has semantic significance.

So much for the hierarchical XOR decomposition of states. The second notion is the AND decomposition, capturing the property that, being in a state, the system must be in *all* of its components. The notation used in statecharts is the physical partitioning of a state box (called **blob** in [H88]) into components, using dashed lines.

Figure 12(i) shows a state Y consisting of AND components A and D, with the property that being in Y entails being in some combination of B or C with E, F or G. We say that Y is the **orthogonal product** of A and D. The components A and D are no different conceptually from any other superstates; they can have defaults, substates, internal transitions, etc. Entering Y from the outside, in the absence of any additional information (like the τ entrance on the right hand side of Fig. 12(ii)), is actually entering the combination (B,F), as a result of the default arrows that lead to B and F. If event α then occurs, it transfers B to C and F to G simultaneously, resulting in the new combined state (C,G). This illustrates a certain kind of **synchronicity**: a single event causing two simultaneous happenings. If, on the other hand, μ occurs at (B, F) it affects the D component only, resulting in (B,E). This, in turn, illustrates a certain kind of **independence**, since the transition is the same

whether, in component A, the system happens to be in B or in C. Both behaviors are part of the orthogonality of A and D, which is the term used in statecharts to describe the AND decomposition. Later we shall discuss the difference between orthogonality and concurrency, or parallelism.

Fig. 12(ii) shows the same orthogonal state Y, but with some transitions to and from it. As mentioned, the τ entrance on the right enters (B,F), but the λ entrance, on the other hand, overrides D's default by entering G directly. But since one cannot be in G alone, the system actually enters the combination (B,G), using A's default. The split ξ entrance on the top of Fig. 12(ii) illustrates an explicit indication as to which combination is to be entered, (B, E) in this case. The γ -event enters a history connector in the area of D, and hence causes entrance to the combination of B (A's default) with the most recently visited state in D. As to the exits in Fig. 12(ii), the ω -event causes an exit from C combined with any of the three substates of D — again a sort of independence property. Had the ω arrow been a merging hyper-edge (like the ξ one, but with the direction reversed) with C and, say, G, as its sources, it would have been a direct specification of an exit from (C,G) only. The most general kind of exit is the one labeled χ on the left hand side of the figure, which causes control to leave AxD unconditionally.

Fig. 13(i) is the conventional AND-free equivalent of Fig. 12(i), and has six states because the components of Fig. 12(i) contain two and three. Clearly, if these had a thousand states each, the resulting "flat" product version would have a million states. This, of course, is the root of the exponential blow-up in the number of states, which occurs when classical finite state automata or state diagrams are used, and orthogonality is our way of avoiding it. (This last comment assumes, of course, that we are specifying using the state-based language alone, not embedded in objects or tasks, etc.) Note that the "in G" condition attached to the β -transition from C in Fig. 12(i) has the obvious consequence in Fig. 13(i): the absence of a β -transition from (C,E). Fig 13(ii) adds to this the ω and χ exiting transitions of Fig. 12(ii), which now show up as rather messy sets of three and six transitions, respectively.

Fig. 14 illustrates the broadcast nature of inter-statechart communication. If after entering the default (B,F,J) event φ occurs, the statechart moves to (C,G,I), since the φ in component H triggered the event α , which causes the simultaneous moves from B to C in component A and from F to G in D. Now, if at the next stage a ψ occurs, I moves back to J, triggering β , which causes a move from C to B, triggering γ , which in turn causes a move from G to F. Thus, ideally in zero time (see Section 10), the statechart goes in this second step from (C,G,I) back to (B,F,J).

As mentioned above, the language has several additional features, though the notions of hierarchy and orthogonality are perhaps its two most significant ones. Besides language features, there are also several interesting semantic issues that arise, such as how to deal with nondeterminism, which hasn't been illustrated here at all, and synchronicity. References [HN89&96, HP91, HK04] have lots of information on these, and Sections 6 and 10 of this paper discuss some of them too.

So much for the basics of the language.

6. Comments on the Underlying Philosophy

When it comes to visuality, encapsulation and side-by-side adjacency are topological notions, just like edge connectivity, and are therefore worthy companions to edges in hierarchical extensions of graphs. Indeed, I believe that topology should be used first when designing a graphical language and only then one should move on to geometry. Topological features are a lot more fundamental than geometric ones, in that topology is a more basic branch of mathematics than geometry in terms of symmetries and mappings. One thing being inside another is more basic than it being smaller or larger than the other, or than one being a rectangle and the other a circle. Being connected to something is more basic than being green or yellow or being drawn with a thick line or with a thin line. I think the brain understands topological features given visually much better than it grasps geometrical ones. The mind can see easily and immediately whether things are connected or not, whether one thing encompasses another, or intersects it, etc. See the discussion on **higraphs** [H88] in Section 8.

Why this emphasis on topology, you may ask? Well, I've always had a (positive) weakness for this beautiful branch of mathematics. I love the idea of an "elastic geometry", if one is allowed a rather crude definition of it; the fact that two things are the same if the one can be stretched and squeezed to become the other. I remember being awed by Brouwer's fixed-point theorem, for example, and the Four-Color problem (in 1976 becoming the Four-Color Theorem). In fact, I started my MSc work in algebraic topology before moving over to theoretical computer science. This early love definitely had an influence on the choices made in designing statecharts.

Statecharts are not exclusively visual/diagrammatic. Their non-visual parts include, for example, the events that cause transitions, the conditions that guard against taking transitions and actions that are to be carried out when a transition is taken. For these, as mentioned earlier, statecharts borrow from both the Moore and the Mealy variants of state machines (see [HU79], in allowing actions on transitions between states or on entrances to or exits from states, as well as conditions that are to hold throughout the time the system is in a state. Which language should be used for these nongraphical elements is an issue we will discuss later.

Of course, the hierarchy and orthogonality constructs are but abbreviations and in principle can be eliminated: Encapsulation can be done away with simply by flattening the hierarchy and writing everything out explicitly on the low level, and orthogonality (as Figs. 12 and 13 show) can be done away with by taking the Cartesian product of the components of the orthogonal parts of the system. This means that these features do not strictly add expressive power to FSMs, so that their value must be assessed by "softer" criteria, such as naturalness and convenience, and also by the size of the description: Orthogonality provides a means for achieving an exponential improvement in succinctness, in both upper- and lower-bound senses [DH88, DH94].

A few words are in line here regarding the essence of orthogonality. Orthogonal state-components in statecharts are *not* the same as concurrent or parallel components of

the system being specified. The intention in having orthogonality in a statechart is not necessarily to represent the different *parts* of the system, but simply to help structure its state space, to help arrange the behavior in portions that are conceptually separate, or independent, or orthogonal. The word 'conceptually' is emphasized here because what counts is whatever is in the mind of the "statifier" — the person carrying out the statechart specification.

This motivation has many ramifications. Some people have complained about the broadcast communication mechanism of statecharts because it's quite obvious that you do not always want to broadcast things to an entire system. One response to this is that we are talking about the mechanism for communication between the orthogonal parts of the statechart, between its "chunks" of state-space, if you will, not between the components — physical or software components — of the actual system. The broadcasting is one of the means for sensing in one part of the state space what is going on in another part. It does not necessarily reflect an actual communication in the real implementation of the system. So, for example, if you want to say on a specification level that the system will only move from state *A* to state *B* if the radar is locked on a target, then that is exactly what you'll say, without having to worry about how state *A* will get to know what the radar is doing. This is true whether or not the other state component actually represents a real thing, such as the radar, or whether it is a non-tangible state chunk, such as whether the aircraft is in air-air mode or in air-ground mode. On this kind of level of abstraction you often really want to be able to sense information about one part of the specification in another, without having to constantly deal with implementation details.

A related response to the broadcasting issue is that no one is encouraged to specify a single statechart for an entire system.² Instead, as discussed in Sections 7 and 9, one is expected to have specified some breakup of the system itself, into functions, tasks, objects, etc., and to have a statechart (or code) for each of these. In this way, the real concurrency, the real separate pieces of the system, occur on a level higher than the statecharts, which in turn are used to specify the behavior of each of these pieces. If within a statechart the behavior is sufficiently complex to warrant orthogonal components, then so be it. In any case, the broadcast mechanism is intended to take effect only within a single statechart, and has nothing to do with the real communication mechanism used for the system itself.

By the way, some of the people who have built tools to support statecharts have chosen to leave orthogonality out of the language altogether, claiming that statecharts don't need concurrency since concurrency is present anyway between the objects of the system... Notable among these is the ObjectTime tool, which later evolved into RoseRT. My own opinion is that orthogonality is probably the most significant and powerful feature of the language, but also the most complex aspect of statecharts to

² This is said despite the fact that in the basic paper on statecharts [H84&87], to be discussed later, I used a single statechart to describe an entire system, the Citizen digital-watch. That was done mainly for presentational reasons.

deal with. So, the decision to leave it out is often made simply to render the task of building a tool that much easier...

Let us return briefly to the two key adjectives used earlier, namely "clear" and "precise", which underlie the choice of the term **visual formalism** [H84&87,H88]. Concerning clarity, the aphorism that a picture is worth a thousand words is something many people would agree with, but it requires caution. Not everything can be visualized; not everything can be depicted visually in a way that is clear and fitting for the brain. (This is related to the discussion above about topology versus geometry.) For some mysterious reason, the basic graphics of statecharts seemed from the very start to vibe well with the avionics engineers at the IAI. They were very fast in grasping the hierarchy and the orthogonality, the high- and low-level transitions and default entries, and so forth.

Interestingly, the same seemed to apply to people from outside our small group. I recall an anecdote from somewhere in late 1983, in which in the midst of one of the sessions at the IAI the blackboard contained a rather complicated statechart that specified the intricate behavior of some portion of the Lavi avionics system. I don't quite remember now, but it was considerably more complicated than the statecharts in Figs. 7–9. There was a knock on the door and in came one of the air force pilots from the headquarters of the project. He was a member of the "customer" requirements team, so he knew all about the intended aircraft (and eventually he would probably be able to fly one pretty easily too...), was smart and intelligent, but he had never seen a state machine or a state diagram before, not to mention a statechart. He stared for a moment at this picture on the blackboard, with its complicated mess of blobs, blobs inside other blobs, colored arrows splitting and merging, etc., and asked "What's that?" One of the members of the team said "Oh, that's the behavior of the so-and-so part of the system, and, by the way, these rounded rectangles are states, and the arrows are transitions between states". And that was all that was said. The pilot stood there studying the blackboard for a minute or two, and then said, "I think you have a mistake down here, this arrow should go over here and not over there"; and he was right.

For me, this little event was significant, as it really seemed to indicate that perhaps what was happening was "right", that maybe this was a good and useful way of doing things. If an outsider could come in, just like that, and be able to grasp something that was pretty complicated but without being exposed to the technical details of the language or the approach, then maybe we are on the right track. Very encouraging.

So much for clarity and visuality. As to precision and formality, later sections discuss semantics and supporting tools in some detail, but for now it suffices to say that one crucial aspect that was central to the development of the language from day one was **executability**. Being able to actually execute the specification of the Lavi's behavior was paramount in my mind, regardless of the form this specification ended up taking. I found it hard to imagine the usefulness of a method for capturing behavior that makes it possible merely to say some things about behavior, to give snippets of the dynamics, observations about what happens or what could happen, or to provide

some disconnected or partially connected pieces of behavior. The whole idea was that if you build a statechart, or a collection of statecharts, everything has to be rigorous enough to be run, to be executable, just like software written in any old (or new...) programming language. Whether the execution is carried out in an interpreter mode or in a compiler mode is a separate issue, one to which we'll return later on. The main thing is that executability was a basic, not-to-be-compromised, underlying concern during the process of designing the language.

This might sound strange to the reader from his/her present vantage point, but back in 1983 system-development tools did not execute models at all, something else we shall return to later. Thus, turning the doodling into a language and then adding features to it had to be done with great care. You had to have a full conception of what each syntactically allowed combination of features in the language means, in terms of how it is to be executed under any set of circumstances.

7. 1984–1986: Building a Tool

Once the basics of the language were there, it seemed natural to want to build a tool that would have the ability not only to draw, or prepare, statecharts but also to execute them. Besides having to deal with the operational semantics of graphical constructs, such a tool would have to deal with the added complication of statecharts over and above classical finite-state machines: A typical snapshot of a statechart in operation contains not just a single state, but rather a vector, or an array, of states, depending on which orthogonal components the chart is in at the moment. Thus, this vector is flexible, given the basic maxim of the language, which is that states can be structured with a mixture of hierarchy and orthogonality and that transitions can go between levels. The very length of the vector of states changes as behavior progresses.

In a discussion with Amir Pnueli in late 1983, we decided to take on a joint PhD student and build a tool to support statecharts and their execution. Amir was, and still is, a dear colleague at the Weizmann Institute and had also been my MSc thesis supervisor 8-9 years earlier. Then, at some point, a friend of ours said something like, "Oh fine, you guys will build your tool in your academic setting, and you'll probably write some nice academic papers about it." And then he added, "You see, if this statechart stuff is just another idea, then whatever you do will not make much difference anyway, but if it has the potential of becoming useful in the real world then someone else is going to build a *commercial* tool around it; they will be the ones who get the credit, they will make the money and the impact, and you guys will be left behind". This caused us to rethink our options, a process that resulted in the founding of a company in Israel in April 1984, by the name of AdCad, Ltd. The two main founders were the brothers Ido and Hagi Lachover, and Amir Pnueli and I joined in too. The other three had previously owned a software company involved in totally different kinds of systems, but in doing so had acquired the needed industrial experience. The company was re-formed in 1987 as a USA entity, called I-Logix,

Inc., and AdCad became its R&D branch, renamed as I-Logix Israel, Ltd.³

By 1986 we had built a tool for statecharts called **Statemate**. At the heart of a Statemate model was a functional decomposition controlled by statecharts. The user could draw the statecharts and the model's other artifacts, could check and analyze them, could produce documents from them, and could manage their configurations and versions. However, most importantly, Statemate could fully execute them. It could also generate from them, automatically, executable code; first in Ada and later also in C.

Among the other central figures during that period were Rivi Sherman and Michal Politi. In fact, it was in extensive discussions with Rivi, Michal and Amir that we were able to figure out how to embed statecharts into the broader framework that would capture the structure and functionality of a large complex system. To this end, we came up with the diagrammatic language that was used in Statemate for the hierarchical functional structuring of the model, which we called **activity-charts**. An activity-chart is an enriched kind of hierarchical data-flow diagram, where the semantics of arrows is the possible flow of information between the incident functions (which are called activities). Each activity could be associated with a controlling statechart (or with code), which would also be responsible for inter-function communication and cooperation. Statemate also enabled you to specify the actual structure of the system, using **module-charts**, which specify the real components in the implementation of the system and their connections. In this way, the tool supported a three-way model-based development framework for systems: structure, functionality and behavior.

Statemate is considered by many to be the first real-world tool to carry out true model executability and full code generation. I think it is not a great exaggeration to claim that the ideas underlying Statemate were really the first serious proposal for **model-driven system development**. These ideas were perhaps somewhat before their time, but were of significance in bringing about the eventual change in attitude that I think permeates modern-day software engineering. The recent UML effort and its standardization by the OMG (see Section 10) can be viewed a subsequent important step in steering software and systems engineering towards model-driven development.

Setting up the links between the statecharts and the activity-charts turned out to be very challenging, requiring among other things that we enrich the events, conditions and actions in the statecharts so that they could relate to the starting and stopping of activities, the use of variables and data types, time-related notions, and much more. After working all this out and completing the first version of Statemate in early 1986, we came across the independent

³ I-Logix survived as a private stand-alone company for 22 long years, amid the many dips in high-tech. In recent years I maintained a very inactive and low-profile connection with the company, until it was acquired by Telelogic in March 2006. As of the time of writing I have no connection with either.

work of Ward and Mellor [WM85] and Hatley and Pirbhai [HP87], who also linked a functional decomposition with a state-based language (theirs was essentially conventional FSMs). It was very satisfying to see that many of the decisions about linking up the two were common to all three approaches. Several years later, Michal Politi and I sat down to write up a detailed report about the entire Statemate language set, which appeared as a lengthy technical report from I-Logix [HP91]. It took several years more for us to turn this into a fully fledged book [HP96].

Statemate had the ability to link the model to a GUI mockup of the system under development (or even to the real system hardware). Executability of the model could be done directly or by using the generated code, and could be carried out in many ways with increasing sophistication. You could execute the model interactively (with the user playing the role of the system's environment), in batch mode (reading in external events from files), or in programmed mode. Just as one example, you could use breakpoints and random events to help set up and control a complex execution from which you could gather the results of interest. In principle, you could thus set Statemate up to "fly the aircraft" for you, and then come in the following day and find out what had happened. See [H92] for a more detailed discussion of model execution possibilities.

During the two years of the development of Statemate, Jonah Lavi from the IAI and his team were also very instrumental. They served as a highly useful beta site for the tool and also participated in making some of the decisions around its development. Jonah's ideas were particularly influential in the decision to have modulecharts be part of Statemate.

Over the years, I-Logix built a number of additional tools, notable among which was a version of Statemate for hardware design, in which the statecharts were translated into a high-level hardware-description language. Much later, in the mid-1990's, we built the **Rhapsody** tool, based on **object-oriented statecharts**, about which we will have more to say in Section 9.

In the early years of I-Logix, I tried hard — but failed — to convince the company's management to produce a cheap (or free) version of Statemate for use by students. My feeling was that students of programming and software engineering should have at their disposal a simple tool for drawing and executing statecharts, connected to a GUI, so that they could build running applications easily using visual formalisms. This could have possibly expedited the acceptance of statecharts in industry. Instead, since Statemate was the only serious statechart tool around but was so very expensive, many small companies, university teachers and students simply couldn't afford it. Things have changed, however, and companies building such tools, including I-Logix, typically have special educational deals and/or simplified versions that can be downloaded free from the internet.

8. The Woes of Publication

In November 1983, I wrote an internal document at the IAI (in Hebrew), titled "Foundations of the State-Based Approach to The Description of System Operation (see Figs. 15-16), which contained an initial account of

statecharts. At the time, my take was that this was but a nice visual way to describe states and transitions of more complex behavior than could be done conveniently with finite-state diagrams. I felt that the consulting job at IAI had indeed been successful, resulting, as it were, in something of use to the engineers of the Lavi avionics project. I had given no thought to whether this was indeed particularly new or novel, believing that anyone seriously working with state machines for real-world systems was probably doing something very similar. It seemed too natural to be new. After all, hierarchy, modularity and separation of concerns were engrained in nearly everything people were writing about and developing for the engineering of large systems.

Then one day, in a routine conversation with Amir Pnueli (this preceded the conversation reported above that resulted in our co-founding AdCad/I-Logix), he asked me, out of curiosity, what exactly I was doing at the Aircraft Industries on Thursdays. So I told him a little about the avionics project and the problem of specifying behavior, and then added something about proposing what seemed to be a rather natural extension of finite-state diagrams. He said that it sounded interesting and asked to see what the diagrams looked like. So I went to the blackboard (actually, a whiteboard; see the photo in Fig.17, taken a few months later) and spent some time showing him statecharts. He said something to the effect that he thought this was nice and interesting, to which I said, "Maybe, but I'm certain that this is how everyone works". He responded by saying that he didn't think so at all and proceeded to elaborate some more. Now, although we were both theoreticians, he had many more years of experience in industry (e.g., as part of the company he was involved in with the Lachover brothers), and what he said seemed convincing. After that meeting it made sense to try to tell the story to a broader audience, so I decided to write a "real" paper and try to get it published in the computer science literature.

The first handwritten version was completed in mid-December of 1983 (see Fig. 18). In this early version the word **statification** was used to denote the process of preparing a statechart description of a system. The paper was typed up, then revised somewhat (including the title) and was distributed as Technical Report CS84-05 of our Department at the Weizmann Institute in February of 1984 [H84&87]; see Fig. 19.

The process leading to the eventual publication of this paper is interesting in its own right. For almost two years, from early 1984 until late 1985, I repeatedly submitted it to what seemed to be the most appropriate widely read venues for such a topic. These were, in order, *Communications of the ACM*, *IEEE Computer* and *IEEE Software*. The paper was rejected from all three of these journals. In fact, from *IEEE Computer* it was rejected twice — once when submitted to a special issue on visual languages and once when submitted as a regular paper. My files contain quite an interesting collection of referee reports and editors' rejection letters. Here are some of the comments therein:

"I find the concept of statecharts to be quite interesting, but unfortunately only to a small segment of our readership. I find the information presented to be somewhat innovative, but not wholly new. I feel that the use of the digital watch

example to be useful, but somewhat simple in light of what our readership would be looking for."

"The basic problem [...] is that [...] the paper does not make a specific contribution in any area."

"A research contribution must contain 'new, novel, basic results'. A reviewer must certify its 'originality, significance, and accuracy'. It must contain 'all technical information required to convince other researchers in the area that the results are valid, verifiable and reproducible'. I believe that you have not satisfied these requirements."

"I think your contribution is similar to earlier contributions."

"The paper is excellent in technical content; however, it is too long and the topic is good only for a very narrow audience."

"I doubt if anyone is going to print something this long."

Indeed, the paper was quite long; it contained almost 50 figures. The main running example was my Citizen Quartz Multi-Alarm digital wristwatch (see Fig. 20), which was claimed in the rejection material by some to be too simple an example for illustrating the concepts and by others to be far too detailed for a scientific article... Some claimed that since the paper was about the much studied finite-state machine formalism it could not contain anything new or interesting...

One must understand that in the mid-1980s there was only scant support for visual languages. Visual programming in the classical sense had not really succeeded; it was very hard to find ways to visualize nontrivial algorithmic systems (as opposed to visualizing the dynamic running of certain algorithms on a particular data structure), and the only visual languages that seemed to be successful in system design were those intended to specify structure rather than behavior. Flowcharts, of course, were a failure in that most people used them to help explain the real thing, which was the computer program. There was precious little real use of flowcharts as a language that people programmed in and then actually executed. In terms of languages for structure, there were structure diagrams and structure charts, hierarchical tree-like diagrams, and so on. The issue of a visual language with precise semantics for specifying behavior was not adequately addressed at all. Petri nets were an exception [R85], but except for some telecommunication applications they did not seem to have caught on widely in the real world. My feeling was that this had mainly to do with the lack of adequate support in Petri nets for hierarchical specification of behavior.

The state of the art on diagrammatic languages at the time can be gleaned from the book by Martin and McClure titled *Diagramming Techniques for Analysts and Programmers* [MM85]. This book discussed many visual techniques, but little attention was given to the need for solid semantics and/or executability. Curiously, this book could have helped convince people that visual languages should *not* be taken seriously as means to actually program a system the way a standard programming language can.

Coming back to the statecharts paper, the inability to get it published was extremely frustrating. Interestingly, during the two years of repeated rejections, new printings of the 1984 technical report had to be prepared, to address the multitude of requests for reprints. This was before the era of the internet and before papers were sent around electronically. So here was a paper that no one wanted to publish but that so many seemed to want to read... I revised the paper twice during that period, and the title changed again, to the final "Statecharts: A Visual Formalism for Complex Systems". Eventually, two and half years later, in July 1987, the paper was published in the theoretical journal *Science of Computer Programming* [H84&87]. That happened as a result of Amir Pnueli, who was one of its editors, seeing the difficulties I was having and soliciting the paper for the journal.⁴

In the revisions of the paper carried out between 1984 and 1987, some small sections and discussions that appeared in earlier versions were removed. One topic that appeared prominently in the original versions and was later toned down, appearing only in a very minimalistic way in the final paper, was the idea of having states contain state protocols, or **statocols**. These were to include information about the behavior that was not present in the charts themselves. The modern term for this kind of information behavior is the **action language**, i.e., the medium in which you write your events, conditions, actions, etc. The question of whether the action language should be part of the specification language itself or should be taken to be a subset of a conventional programming language is the subject of a rather heated debate that we will return to later.

A few additional papers on statecharts were written between 1984 and 1987. One was the paper written jointly with Pnueli on **reactive systems** [HP85]. It was born during a plane trip that we took together, flying to or from some conference. We were discussing the special nature of the kinds of systems for which languages like statecharts seemed particularly appropriate. At some point I complained about the lack of a special term to denote them, to which he responded by saying he thought such systems should be termed reactive. "Bingo", I said, "we have a new term"! Interestingly, this paper (which also contained a few sections describing statecharts) was written about two years after the original statecharts paper, but was published (in a NATO conference proceedings) a year earlier...

Another paper written during that period was actually published without any trouble at all, in the *Communications of the ACM* [H88]. It concentrates on the graphical properties of the statecharts language, disregarding the intended semantics of nodes as dynamic

⁴ A note on the title page of the published version states that the paper was "Received December 1984, Revised July 1986". The first of these is an error – probably made in the typesetting stage – since submission to Pnueli was made in December 1985. By the way, this story of the repeated rejections of the paper would not be as interesting were it not for the fact that in the 20 years or so since its publication it has become quite popular. According to Citeseer, it has been for several years among the top handful of most widely quoted papers in computer science, measured by accumulated citations since publication (in late 2002 it was in the second place on the list).

states and edges as transitions. The paper defined a **higraph** to be the graphical artifact that relates to a directed graph just as a statechart relates to a finite-state diagram.⁵ The idea was to capture the notions of hierarchy, orthogonality, multilevel and multinode transitions, hyperedges, and so on, in a pure set-theoretic framework. It too contains material on statecharts (and a simplified version of the digital-watch example) and since it appeared in a journal with a far broader readership than *Science of Computer Programming* it is often used as the de facto reference to statecharts.

The third paper that should be mentioned here was on the semantics of statecharts [HPSR87]. It was written jointly with the "semantics group" — the people involved in devising the best way to implement statecharts in Statemate — and provided the first formal semantics of the language. However, some of the basic decisions we made in that paper were later changed in the design of the tool, as discussed in Section 10.

Another paper was the one written on the Statemate tool itself [H+88&90], co-authored by the entire Statemate team at Ad-Cad/I-Logix. Its appeal, I think, goes beyond the technical issue of showing how statecharts can be implemented (the lack of which several of the referees of the basic statecharts paper complained about). In retrospect, as mentioned earlier, it set the stage for and showed the feasibility of the far broader concepts of **model-driven development**, true **model executability** and **full code generation**. These are elaborated upon in a later, more philosophical paper, "*Biting the Silver Bullet*" [H92], which also contained a rebuttal of Fred Brooks' famous "*No Silver Bullet*" paper [B87].

To close this section, an unfortunate miscalculation with regards to publication should be admitted. This was my failure to write a book about the language early on. As mentioned in the previous section, it was only in 1996 that the book with Michal Politi on Statemate was published [HP96]. This was definitely a mistake. I did not realize that most engineers out there in the real world rarely have the time or inclination to read papers, and even if they do they very rarely take something in a paper seriously enough to become part of their day-to-day practice. One has to write a book, a popular book. I should have written a technical book on statecharts, discussing the language in detail, using many examples, describing the tool we already had that supported it, and carefully working out and describing the semantics too. This could have helped expose the language to a broader audience a lot earlier.

9. 1994–1996: The Object-Oriented Version

In the early 1990s Eran Gery, who at the time was (and still is) one of the key technical people at I-Logix, became very interested in object-oriented modeling. As it turned out, several people, including Jim Rumbaugh and

Grady Booch, had written about the use of statecharts in object-oriented analysis and design; see, e.g., [B94, RBPEL91, SGW94]. It was Eran's opinion that their work left some issues that still had to be dealt with in more detail; for example, the semantics of statecharts were not worked out properly, as were the details of some of the dynamic connections with the objects. Also, they had not built a tool such as Statemate for this particular, more modern, OO approach. In the terminology of the present paper, their version of the language was not (yet) executable.

Despite being well aware of object-oriented programming and the OO programming languages that existed at the time, I was not as interested in or as familiar with this work on OO modeling as was Eran. Once Statemate had been designed and its initial versions built, the implementational issues that arose were being dealt with adequately by the I-Logix people, and I was spending most of my time on other topics of research. Eran did some gentle prodding to get me involved, and we ended up taking a much closer look at the work of Booch, Rumbaugh and others. This culminated in a 1996 paper, "*Executable Object Modeling with Statecharts*", in which we defined **object-oriented statecharts**, an OO version of the language, and worked out the way we felt the statecharts should be linked up with objects and executed [HG96&97]. One particular issue was the need for two modes of communication between objects, direct synchronous invocation of methods and asynchronous queued events. There were also many other aspects to be carefully thought out that were special to the world of objects, such as the creation and destroying of objects and multithreaded execution. The main structuring mechanism is that of a class in a class diagram (or an object instance in an object model diagram), each of which can be associated with a statechart. A new copy of the statechart is spawned whenever a new instance of the class is created. See Fig. 21 for two examples of statecharts taken from that paper.

In the paper we also outlined a new tool for supporting all of this, which I-Logix promptly started to build, called **Rhapsody**. Eran championed and led the entire Rhapsody development effort at I-Logix, and he still does.

And so we now have two basic tools for statecharts — Statemate, which is not object-oriented and is intended more for systems people and for mixed hardware/software systems, and Rhapsody, which is intended more for software systems and is object-oriented in nature. One important difference between the tools, which we shall elaborate upon in Section 10, is that the semantics of statecharts in Statemate is synchronous and in Rhapsody it is, by and large, asynchronous. Another subtle but significant difference is reflected in the fact that Statemate was set up to execute statecharts directly, in an interpreter mode that is separate from the code generator. In contrast, the model execution in Rhapsody is carried out solely by running the code generated from the model. Thus, Rhapsody could be thought of as representing a high-level programming language that is compiled down into runnable code. Except, of course, that the statechart language is a level higher than classical programming languages, in that the translation from it was made into

⁵ A sub-formalism of higraphs, which contains hierarchy and multi-level transitions has been called **compound graphs** [MS88,SM91].

C++, Java or C, etc. Another important difference is that a decision was made to make the action language of Rhapsody be a subset of the target programming language. So you would draw statecharts in Rhapsody and the events and actions specified along transitions and in states, etc., are fragments of, say, C++ or Java. (The action language in Fig. 21, for example, is C++.) These differences really turn Rhapsody into more of a high-level programming tool than a system-development tool. See also the discussion on the UML in Section 10.

There are now several companies that build tools that support statecharts. There are also many variants of the language. One of the most notable early tools is **ObjecTime**, built by Bran Selic and Paul Ward and others. This tool later became **RoseRT**, from Rational Corp. **StateRover** is another statechart tool, built by my former student, Doron Drusinsky. Finally, **Stateflow** is a statechart tool embedded in Matlab (which is used widely by people interested in control systems); its statecharts can be effortlessly linked to Matlab's other modeling and analysis tools.

It is worth viewing the implementation issue in a slightly broader perspective. In the early 1980s, essentially none of the tools offered for system development using graphical languages were able to execute models or generate running code. If one wants to be a bit sarcastic about it, these so-called CASE tools (the acronym standing for computer-aided software engineering) were not much more than good graphic editors, document generators, configuration managers, etc. It would not be much of an exaggeration to say that pre-1986 modeling tools were reminiscent of support tools for a programming language with lot of nice features but with no compiler (or interpreter). You could write programs, you could look at them, you could print them out, you could ask all kind of queries such as "list all the integer variables starting with D", you could produce documents, you could do automatic indentation, and many other niceties; everything except run the programs!

Of course, in the world of complex systems, tools that do these kinds of things – checking for the consistency of levels and other issues related to the validity of the syntax, offering nice graphic abilities for drawing and viewing the diagrams, automatically generating documents according to pre-conceived standards, and so on – are very important. Although these features are crucial for the process of building a large complex system, I was opposed to the hype and excitement that in pre-1986 years tended to surround such tools. My take was that the basic requirement of a tool for developing systems that are dynamic in nature is the ability not only to describe the behavior, but also to analyze it and execute it dynamically. This philosophy underlies the notion of a visual formalism, where the language is to be both diagrammatic and intuitive in nature, but also mathematically rigorous, with a well-defined semantics sufficient to enable tools to be built around it that can carry out dynamic analysis, full model execution and the automatic generation of running code; see [H92].

10. On Semantics

It is worth dwelling on the issue of **semantics** of statecharts. In a letter from Tony Hoare after he read the 1984 technical report on statecharts, he said very simply that the language "badly needs a semantics". He was right. I was overly naïve at the time, figuring that writing a paper that explained the basics of the language's operation and then building a tool that executes statecharts and generates code from them would be enough. This approach took its cue from programming language research, of course, where people invent languages, describe them in the literature and then build compilers for them. That this was naïve is a consequence of the fact that there are several very subtle and slippery issues around the semantics of any concurrent language – statecharts included. These not only have to be decided upon when one builds a tool, something we actually took great pains to do properly when designing Statemate, but they also have to be written up properly for the scientific community involved in the semantics of languages.

In retrospect, what we didn't fully realize in those early years was how different statecharts were from previous specification languages for real-time embedded systems – for better or for worse. We knew that the language had to be both executable and easily understandable by many different kinds of people who hadn't received any training in formal semantics. But at the same time, as a team wanting to build a tool, we also had to demonstrate quickly to our sponsors, the first being IAI, that ours was an economically viable idea; so we were under rather great time pressure. Due to the high level of abstraction of statecharts, we had to resolve several rather deep semantical problems that apparently hadn't been considered before in the literature, at least not in the context of building a real-world tool intended for large and complex systems. What we didn't know was that some of these very issues were being investigated independently, around the same time, by various leading French researchers, including Gérard Berry, Nicholas Halbwachs and Paul le Guernic (who later coined the French phrase *L'approche synchrone*, "the synchronous approach", for their kind of work).

In actuality, during the 1984-6 period of designing Statemate, we did not do such a clean and swift job of deciding on the semantics. We had dilemmas regarding several semantic issues, a couple of which were particularly crucial and central. One had to do with whether a step of the system should take zero time or more, and another had to do with whether the effects of a step are calculated and applied in a fixpoint-like manner in the same step, or are to take effect only in the following one. The two issues are essentially independent; one can adopt any of the four combinations. Here is not the proper place to explain the subtlety of the differences, but the first issue, for example, has to do with whether or not you adopt the pure **synchrony hypothesis**, generally attributed to Berry, whereby steps take zero time [BG92]. Of course, these questions have many consequences in terms of how the language operates, whether events can interfere with chain reactions triggered by other events, how time itself is modeled, and how time interleaves with the discrete event dynamics of the system.

During that period the main people who were sitting around the table discussing this were Amir Pnueli, Rivi Sherman, Janette Schmidt, Michal Politi and myself, and for some time we used the code names Semantics A and B for the two main approaches we were seriously considering. Both semantics were synchronous in the sense of [BG92] and differed mainly in the second issue above. The paper we published in 1987 was based on Semantics B [HPSR87], but we later adopted semantics A for the Statemate tool itself, which was rather confusing to people coming from outside of our group. In 1989, Amnon Naamad and I wrote a technical report that described the semantics adopted in Statemate [HN89&96], i.e., Semantics A, where the effects of a step are accumulated and are then carried out in the following step. At the time, we did not think that this report was worth publishing — naïveté again — so for several years it remained an internal I-Logix document.

In any case, the statecharts of Statemate really constitute a **synchronous language** [B+03], and in that respect they are similar to other, non visual languages in this family, such as Berry's Esterel [BG92], Lustre [CPHP87] and Signal [BG90].

At that time, a number of other researchers started to look at statechart semantics, some being motivated by our own ambivalence about the issue and by the fact that the implemented semantics was not published and hence not known outside the Statemate circle. For example, in an attempt to evaluate the different semantics for statecharts, Huizing, Gerth and de Roever proved one of them to have the desirable property of being fully abstract [HGdR88]. As the years went by, many people defined variants of the statechart language, sometimes dropping orthogonality, which they deemed complicated, and often adding some features or making certain modifications. There were also several papers published that attempted to provide formal, machine-readable semantics for the language, and others that described other tools built around variants thereof.

An attempt to summarize the situation was carried out by von der Beeck, who tried to put some order into the multitude of semantics of statecharts that were being published. The resulting paper [vB94] claimed implicitly that statecharts is not really a well-defined language because of these many different semantics (it listed about twenty such). Interestingly, while [vB94] reported on the many variants of the language with the many varying semantics, it did not report on what should probably have been considered at the time the "official" semantics of the language. This is the semantics we defined and adopted in 1986-7 when building Statemate [HN89&96]; the one I talked about and demonstrated in countless lectures and presentations in the preceding 8 years, but, unfortunately, the only one not published at the time in the widely-accessible open literature...

Around the same time another paper was published, by Nancy Leveson and her team [LHHR94], in which they took a close look at yet another statecharts semantics paper, written by Pnueli and Shalev [PS91]. The Pnueli/Shalev paper provided a denotational fixpoint semantics for statecharts and elegantly showed its equivalence to a certain operational semantics of the language. Leveson and her group did not look at the Statemate tool either and, like von der Beeck, had not seen our then-unpublished

technical report [HN89&96]. The Leveson et al paper was very critical of statecharts, going so far as to hint that the language is unsafe and should not be used, the criticism being based to a large extent on anomalies that they claimed could surface in systems based on the semantics of [PS91].

It seems clear that had a good book about statecharts been available early on, including its semantics and its Statemate implementation, some of this could have been avoided. At the very least we should have published the report on the Statemate semantics. It was only after seeing [vB94, LHHR94] and becoming rather alarmed by the results of our procrastination that we did just that, and the paper was finally published in 1996 [HN89&96].

As to the semantic issues themselves, far more important than the differences between the variants of pre-OO statecharts themselves, as reported upon in [vB94], is the difference between the non-object-oriented and the object-oriented versions of the language, as discussed above. The main semantic difference is in synchronicity. Statemate statecharts, i.e., the version of the language based on functional decomposition, is a synchronous language, whereas Rhapsody statecharts, i.e., the object-oriented version thereof, is an asynchronous one. There are other substantial differences in modes of communication between objects, and there are the issues that arise from the presence of dynamic objects and their creation and destruction, inheritance, object composition, multithreading, and on and on. All these have to be dealt with when one devises an object-oriented version of such a language and builds a tool like Rhapsody, which supports both the objects and their structure and the statecharts and code that drive their behavior.

In the object-oriented realm, a similar publication sin was committed, waiting far too long to publish the semantics of statecharts in Rhapsody. Only very recently, together with Hillel Kugler, did we finally publish a paper (analogous to [HN89&96]) that gives the semantics of statecharts as adopted in Rhapsody and describes the differences between these two subtly different versions of the language [HK04].

This section on semantics cannot be completed without mentioning the **unified modeling language**, the **UML**; see [RJB99, UML]. As the reader probably well knows, Rumbaugh and Booch, together with Ivar Jacobson, got together to form the technical core team of the impressive UML effort, which was later standardized by the object management group (OMG). Although the UML features many graphical languages, many of them have not been endowed with satisfactorily rigorous semantics. The heart of the UML — what many people refer to as its driving behavioral kernel — is the (object-oriented variant of) statecharts language; see Section 9. In the late 1990s Eran Gery and I took part in helping this team define the intended meaning of statecharts in the UML. This had the effect of making UML statecharts very similar to what we had already implemented in Rhapsody.

In fact, currently the two main executable tools for UML-based languages are Rhapsody and RoseRT; the latter, as mentioned above, is a derivative of the earlier ObjecTime tool, and implements a sublanguage of statecharts: for example, it does not support orthogonal

state components.⁶ There are other differences between these two tools that the present paper cannot cover. Also, the issue of whether the action language should be the target programming language, as in Rhapsody, or whether there should be an autonomous action language is still raging in full force and the UML jury is not yet in on this issue.

See the recent [HR04], with its whimsical title "*What's the Semantics of 'Semantics'?*"", for a manifesto about the subtle issues involved in defining the semantics of languages for reactive systems, with special emphasis put on the UML.

11. Biological Modeling with Statecharts

In terms of usage of statecharts, the language appears to be used very widely in computer embedded and interactive systems, e.g., in the aerospace and automotive industries, in telecommunication and medical instrumentation, in control systems, and so on. However, one of the more interesting developments involves statecharts also being used in non-conventional areas, such as modeling biological systems and health-care processes.

Starting in the mid-1980s I had often claimed that biological systems should be viewed as systems the way we know them in the world of computing, and **biological modeling** should be attempted using languages and tools constructed for reactive systems, such as statecharts. One modest attempt to do so was made by a student in our department, Billie Sandak, around 1989. This work was not carried out to completion, and the topic was picked up about ten years later by another student, Naaman Kam, co-supervised by Irun Cohen, a colleague of mine from the Weizmann Institute's Immunology department. The resulting work (written up in [KCH01]) started a flurry of activity, and by now several serious efforts have been made on using statecharts to model biological systems. This includes one particularly extensive effort of modeling T cell development in the thymus gland, done with our student Sol Efroni [EHC03], and others involving, e.g., the pancreas [SeCH06] and the lymph node [SwCH06]. The thymus model, for example, contains many thousands of complex objects, each controlled by a very large and complicated statechart, and has resulted in the discovery of several properties of the system in question; see the recent [EHC07]. Figs. 22 and 23 show, respectively, the front-end of this model and a schematic rendition of parts of the statechart of a single cell. Figs. 24 and 25 show more detailed parts of some of the statecharts from the thymus model during execution.

One of the notions that we came up with during our work on the thymus model is **reactive animation** [EHC05]. The idea is to be able to specify systems for which the front end requires something more than a GUI — specifically, systems that require true animation. A good example would be a traffic or radar system with many

⁶ By the way, Rational's most popular tool, Rational Rose, cannot execute models or produce executable code. In that respect it suffers from the same central weakness afflicting the pre-1986 CASE tools.

elements and targets, such as cars or aircraft, being created, moving in and out of the scene, traveling around, growing and shrinking in size, changing and getting destroyed, etc. Under normal circumstances, this kind of system would have to be programmed using the script language supported by an animation system. Reactive animation allows one to use a state-of-the-art reactive system tool, such as Statemate or Rhapsody, linked up directly and smoothly with an animation tool. The T cell model of [EHC03, EHC07] was built using statecharts in Rhapsody, linked up with the Flash animation tool, and the two work together very nicely. Reactive animation is used extensively also in the pancreas and lymph node models [SeCH06, SwCH06].

12. Miscellaneous

This section discusses some related topics that came up over the years. One is the notion of **overlapping states**, whereby you want the and/or state hierarchy in statecharts to be a directed graph, not a tree. This possibility, and the motivation for it, was already mentioned in the earliest documents on statecharts; see Fig. 26. In work with an MSc student, H.-A. Kahana, the details of how overlapping could be defined were worked out [HK92]. We found that the issue was pretty complicated since, e.g., overlapping can be intermixed not only with the substate facet of the hierarchy but also with orthogonal components. We actually concluded that the complications might outweigh the benefits of implementing the feature.

Although the basic idea is very natural, it appears that such an extension is not yet supported in any of the statechart tools. Incidentally, this does not prevent people from thinking that overlapping is a simple matter, since it is tempting to think only of simple cases, like that of Fig. 26. Some people have approached me and asked "Why doesn't your tool allow me to draw one state overlapping the other? Why don't you simply tell it not to give me the error message when I try to do this in the graphic editor?" Of course, underlying such questions is the naive assumption that if you can draw a picture of something, and it seems to make sense to you, then there is no problem making it part of the language... I often use this exchange to illustrate the difference between what many people expect of a visual language and what a real visual formalism is all about; see the discussion on "the doodling phenomenon" in [HR04].

An additional topic is that of **hybrid systems**. It is very natural to want to model systems that have both discrete and continuous aspects to them. In discussions and presentations on statecharts in the 1980s, I often talked about the possibility of using techniques from control theory and differential equations to model the activities occurring within states in a statechart, but never actually did any work on the idea. Many years later the notion of a hybrid (discrete and continuous) system was put forward by several people, and today there is an active community doing deep research in the area. Many models of hybrid systems are essentially finite-state machines, often rendered using statecharts that are intermixed with techniques for specifying continuous aspects of a system, such as various kinds of differential equations.

The other idea I have been trying to peddle for years but have done nothing much about is to exploit the

structure of the behavior given in statecharts to aid in the **verification** of the modeled system. The philosophy behind this is as follows. We all know that verification is hard, yet there are techniques that work pretty well in practice, such as those based on model checking. However, common verification techniques do not exploit the hierarchical structure or modularity that such models very often have. Now, assume that someone has already made the effort of preparing a statechart-based description of a complex system, and has gone to great pains in order to structure the statecharts nicely to form a hierarchy with multilevel orthogonal components. There should probably be a way to exploit the reasons for the decisions made in this structuring process in carrying out the verification. Perhaps the way to do it is to try to get more information from the "statifier", i.e., the person preparing the statecharts, about the knowledge he or she used in the structuring. For example, just as we expect someone writing a program with a loop to be able to say more about the invariant and convergent properties of that loop, so should we expect someone breaking a system's state space into orthogonal components, or deciding to have a high-level state encompass several low-level states, to be able to say something about the independent or common properties of these pieces of the behavior.

There has actually been quite a lot of work on the verification (especially model checking) of hierarchical state machines, and the availability of various theoretical results on the possibility (or lack thereof) of obtaining significant savings in the complexity of verifying concurrent state machines. There are also some tools that can model-check statecharts. However, my feeling is that the jury is not in yet regarding whether one can adequately formalize this user-provided information and use it beneficially in the verification process.

Finally, I should mention briefly the more recent work with colleagues and students, which can be viewed as another approach, to visual formalisms for complex systems. It has to do with **scenario-based** specification. The statechart approach is **intra-object**, in that ultimately the recommendation is to prepare a statechart for each object of the system (or for each task, function, component, etc., whatever artifacts your system will be composed of). Of course, the statecharts are to also contain information about the communication between the objects, and one could build special controlling statecharts to concentrate on these aspects; however, by and large, the idea of finite-state machines in general, and statecharts in particular, is to provide a way for specifying the behavior of the system per object in an intra-object fashion. The more recent work has to do with scenario-based, **inter-object** specification. The idea is to concentrate on specifying the behavior between and among the objects (or tasks, functions, components, etc.). The main lingua franca for describing the behavior of the system would have to be a language for specifying communication and collaboration between the objects. This became feasible with the advent of **live sequence charts** (or **LSCs**, for short) in work joint with Werner Damm in 1999; see [DH99&01]. Later, with my student Rami Marely, a means for specifying such behavior directly from a GUI was worked out, called **play-in**, as well as a

means for executing the behavior, called **play-out**, and the entire setup and associated methods have been implemented in a tool called the **Play-Engine**; see [HM03].

We have also built mechanisms to bridge between the two approaches, so that one can connect one or more Play-Engines with other tools, such as Rhapsody (see [BHM04]). In this way, one can specify part of the behavior of the system by sequence charts in a scenario-based, inter-object, fashion, and other objects can be specified using statecharts, or even code, in an intra-object fashion.

13. Conclusions

In summary, it would seem that one of the most interesting aspects of this story of statecharts in the making is in the fact that the work was not done by an academic researcher sitting in his/her ivory tower, inventing something and trying to push it down the engineers' throats. Rather, it was done by going into the lion's den, so to speak, working in industry and with the people in industry. This is consistent with the saying that "the proof of the pudding is in the eating".

Other things crucial to the success of a language and an approach to system-development are good supporting tools and excellent colleagues. In my own personal case, both the IAI engineers and the teams at AdCad/I-Logix who implemented the Statemate tool and then the Rhapsody tool were an essential and crucial part of the work. And, of course, a tremendous amount of luck is necessary, especially, as in this case, when the ideas themselves are not that deep and not that technically difficult.

I still believe that almost anyone could have come up with statecharts, given the right background, being exposed to the right kinds of problems and being surrounded by the right kinds of people.

Acknowledgments

Extensive thanks are due to my many colleagues at the Israel Aircraft Industries, at Ad-Cad/I-Logix and at The Weizmann Institute. Some of these have been mentioned and credited in the text itself, but I'd like to express particularly deep gratitude to Jonah Lavi, Amir Pnueli, Eran Gery, Rivi Sherman and Michal Politi. In addition, Moshe Vardi, Willem de Roever and the HOPL III referees made valuable comments on early versions of the paper. The process of writing of this paper was supported in part by the John von Neumann Center for the Development of Reactive Systems at the Weizmann Institute, and by grants from Minerva and the Israel Science Foundation.

(Note: the References section appears after the figures)

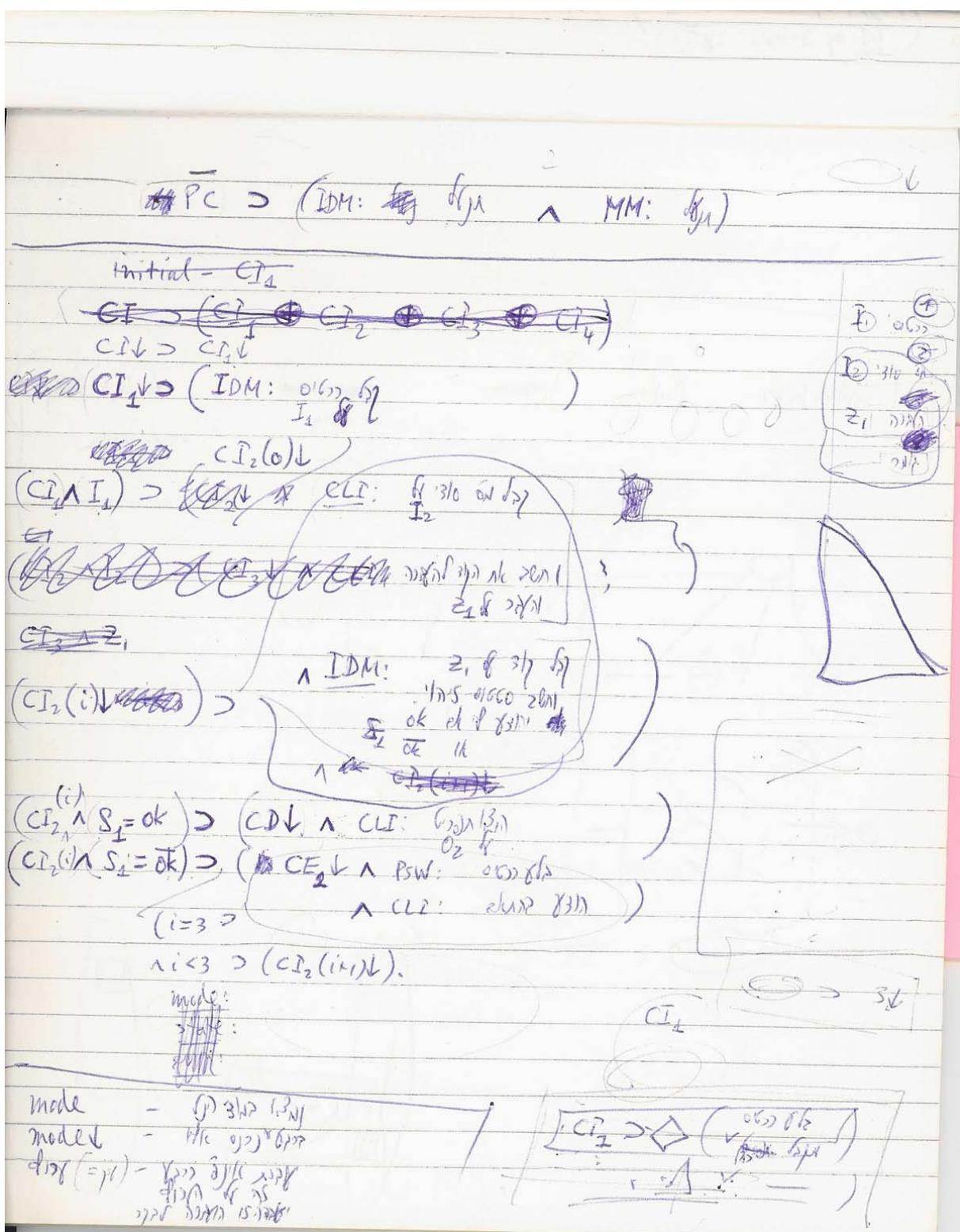


Figure 1. Page from the IAI notes (early 1983; with some Hebrew) showing the first attempt at helping specify the Lavi avionics, using a temporal logic-like formalization.

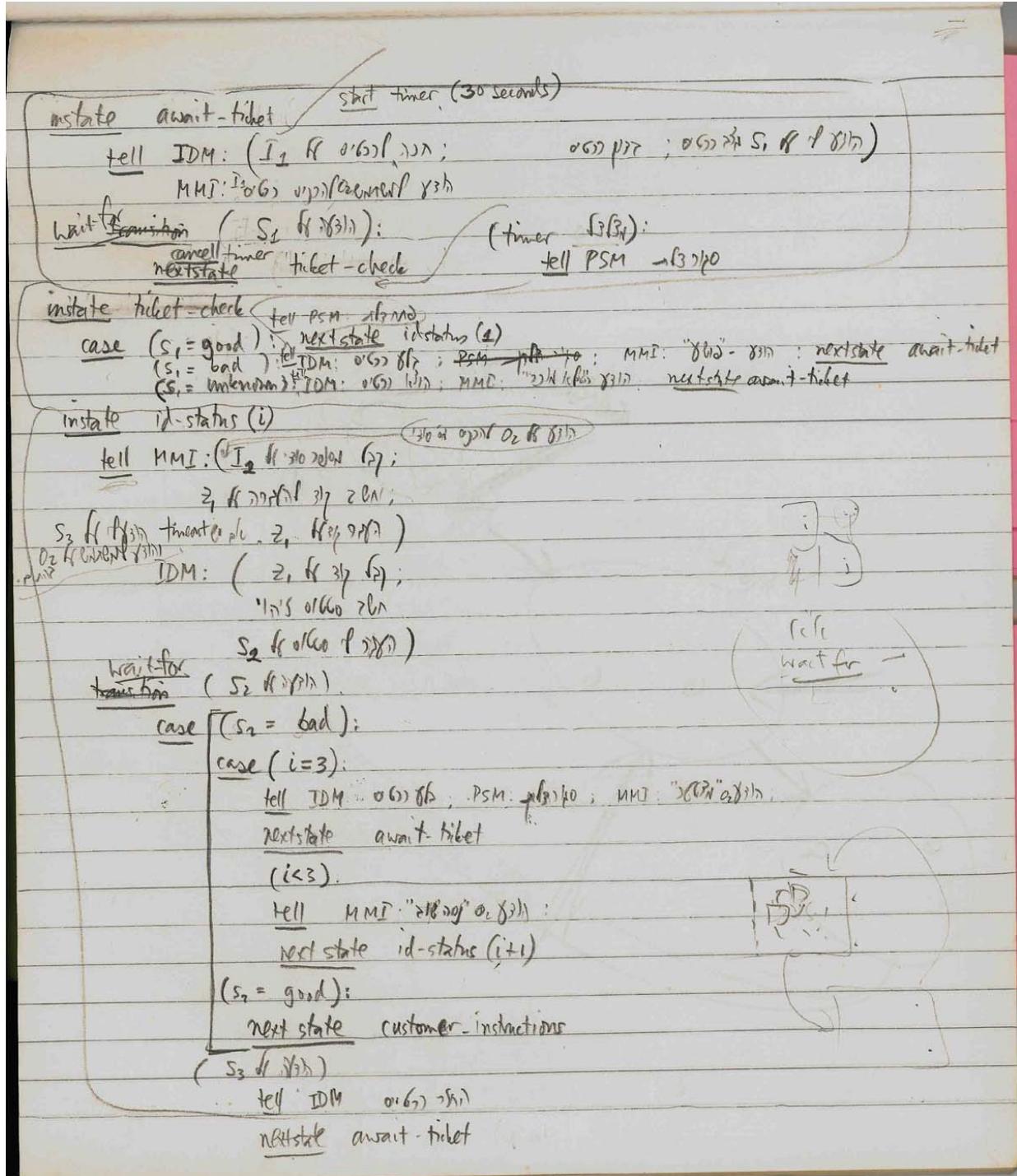


Figure 2. Page from the IAI notes (early 1983; with some Hebrew) showing parts of the Lavi avionics behavior using "statocols", the second attempt — a kind of structured state-transition protocol language. Note the graphical "doodling" on the right hand side, which was done to help clarify things to the engineers, and which quickly evolved into statecharts.

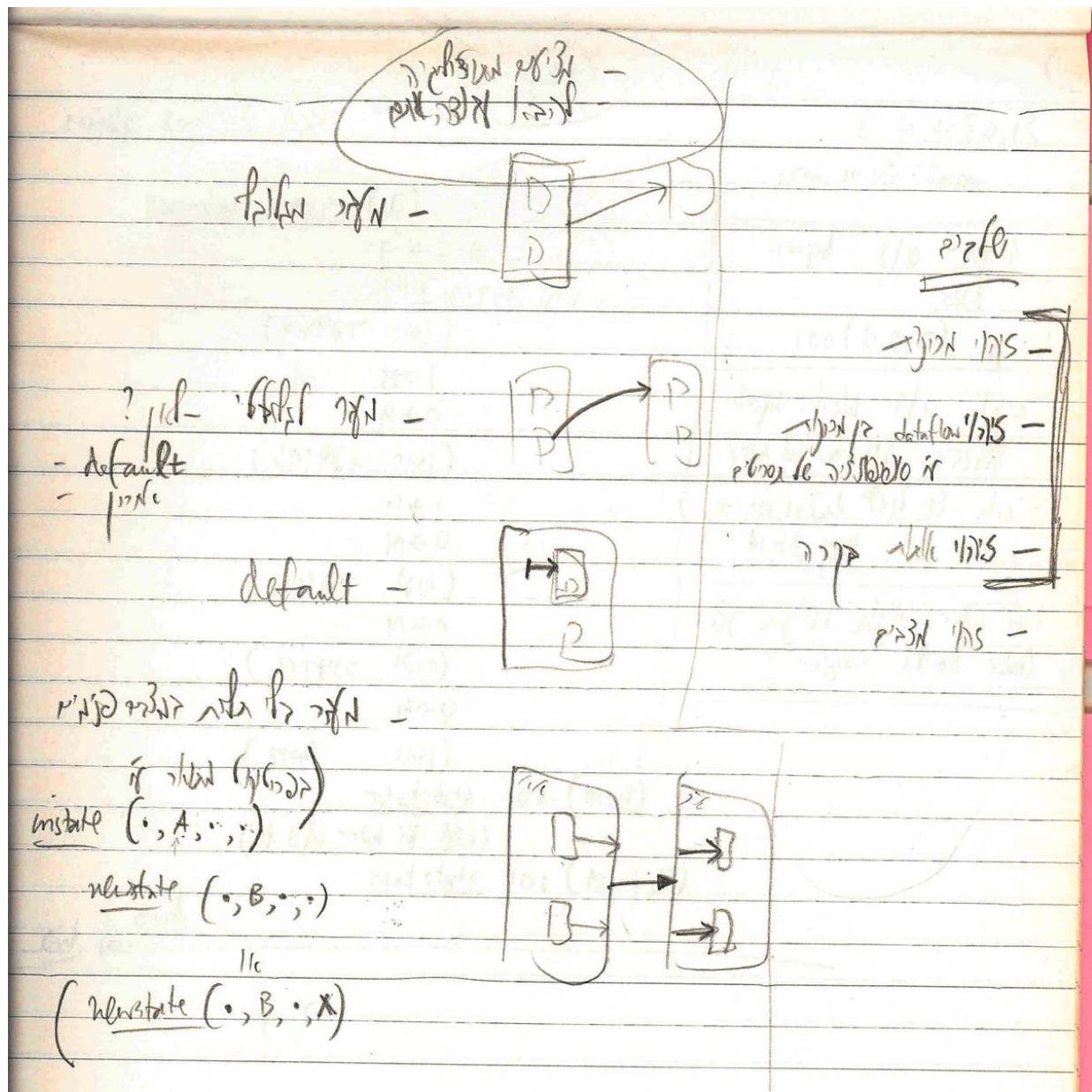


Figure 3. Page from the IAI notes (mid-1983; in Hebrew) showing a first attempt at deciding on graphical/topological elements to be used in the hierarchy of states. Note the use of the term *default* as a generalization to hierarchical states of the notion of a start state from automata theory.

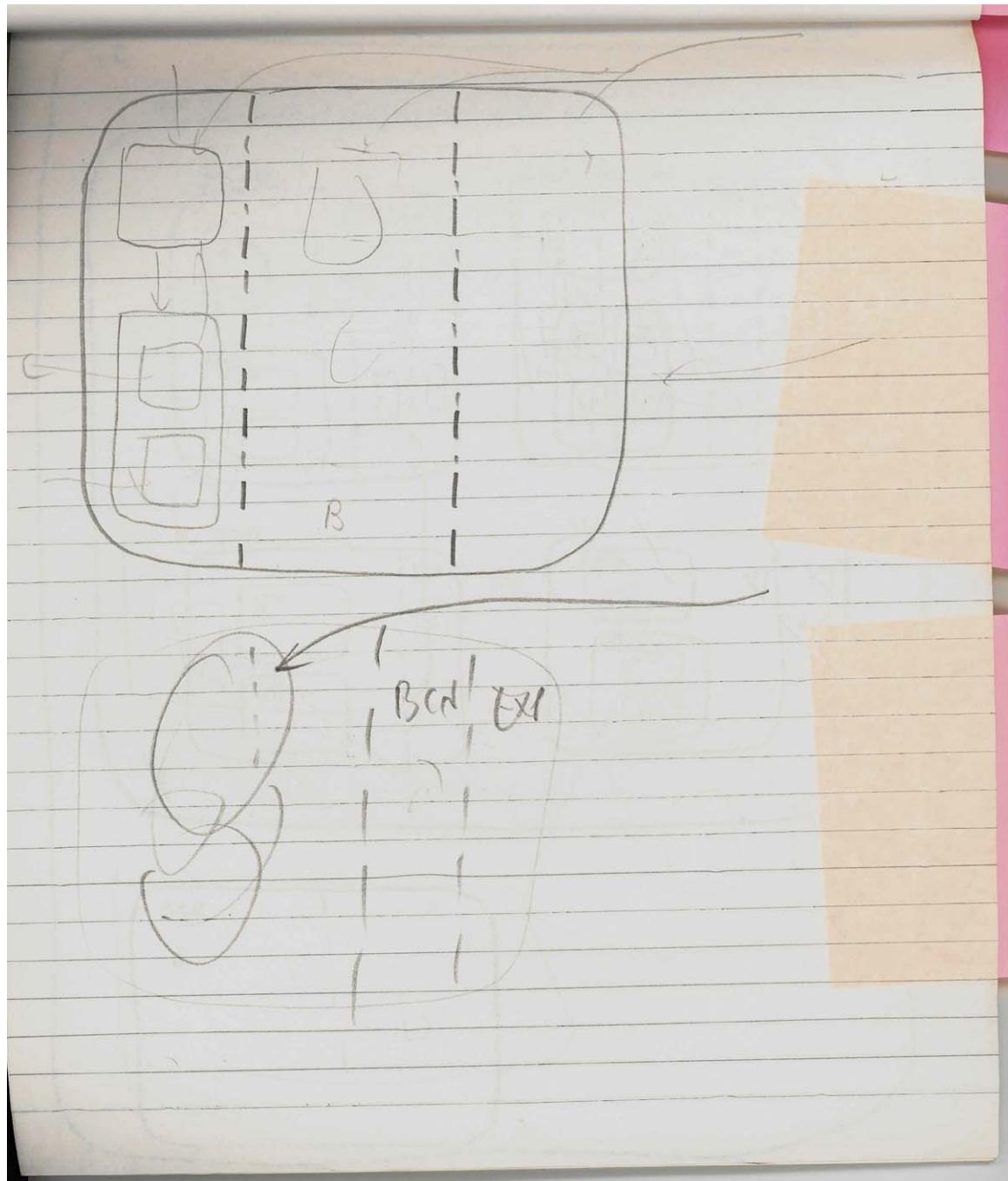


Figure 4. Page from the IAI notes (mid-1983) showing the first rendition of orthogonal state components. Note the hesitation about what style of separation lines to use.

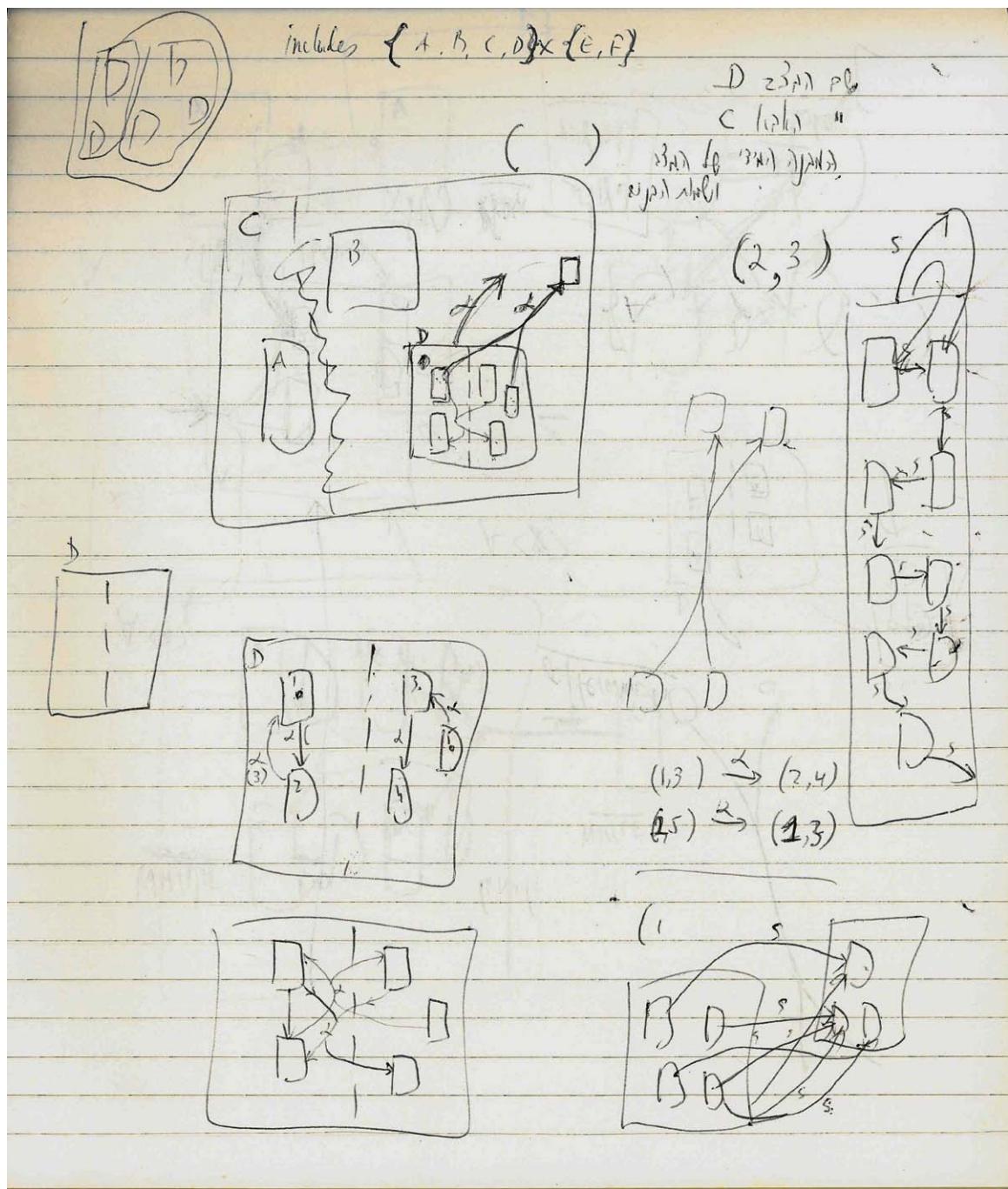


Figure 5. Page from the IAI notes (mid-1983). Constructs shown include hyper-edges, nested orthogonality, transitions that reset a collection of states (chart on right). Note the use of Cartesian products of sets of states (top) to capture the meaning of orthogonality, and the straightforward algebraic notation for transitions between state vectors (bottom third of page, on right).

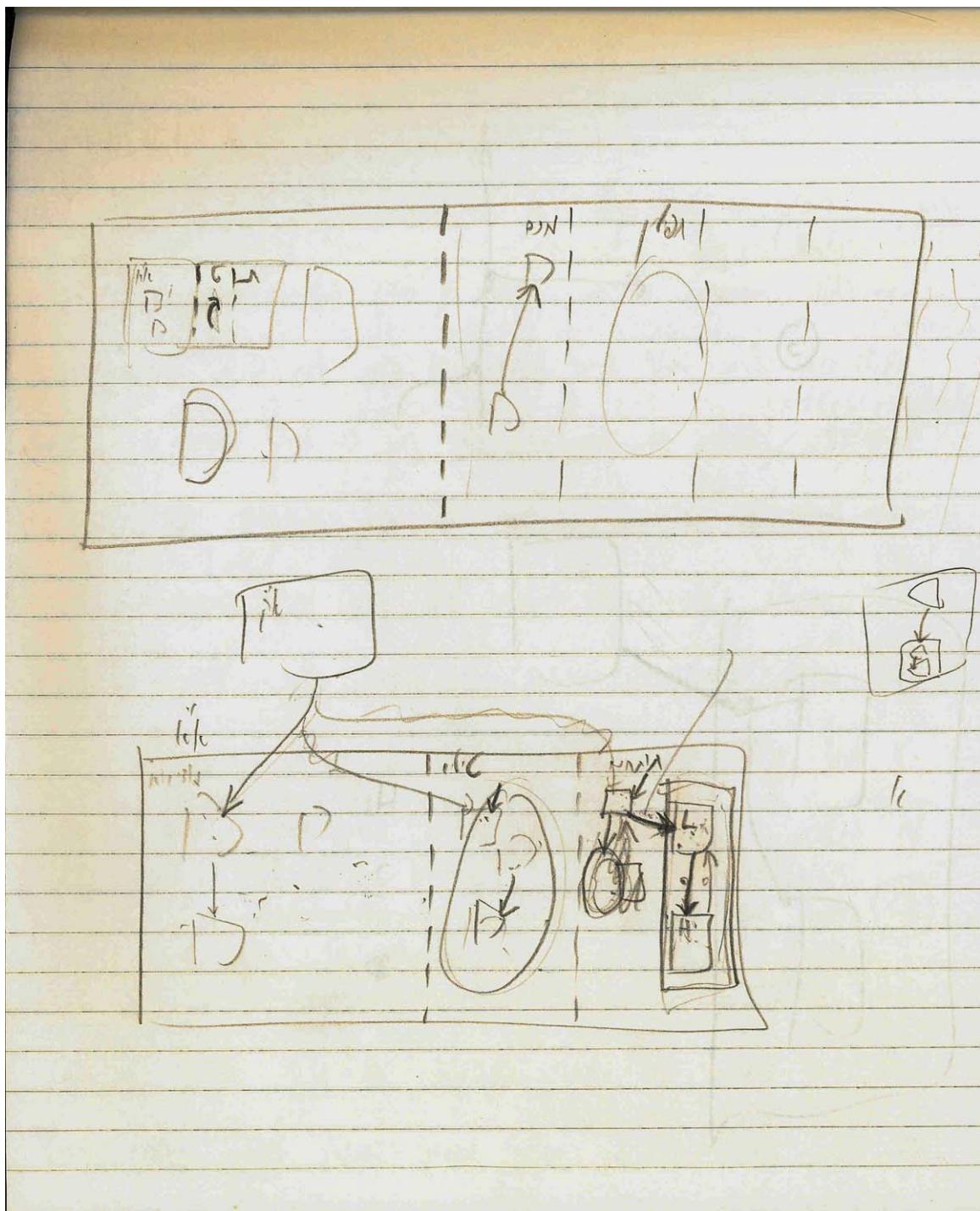


Figure 6. Page from the IAI notes (mid-1983) showing some initial statechart attempts for the Lavi avionics. Note the nested orthogonality (top left) and the inter-level transitions (bottom).

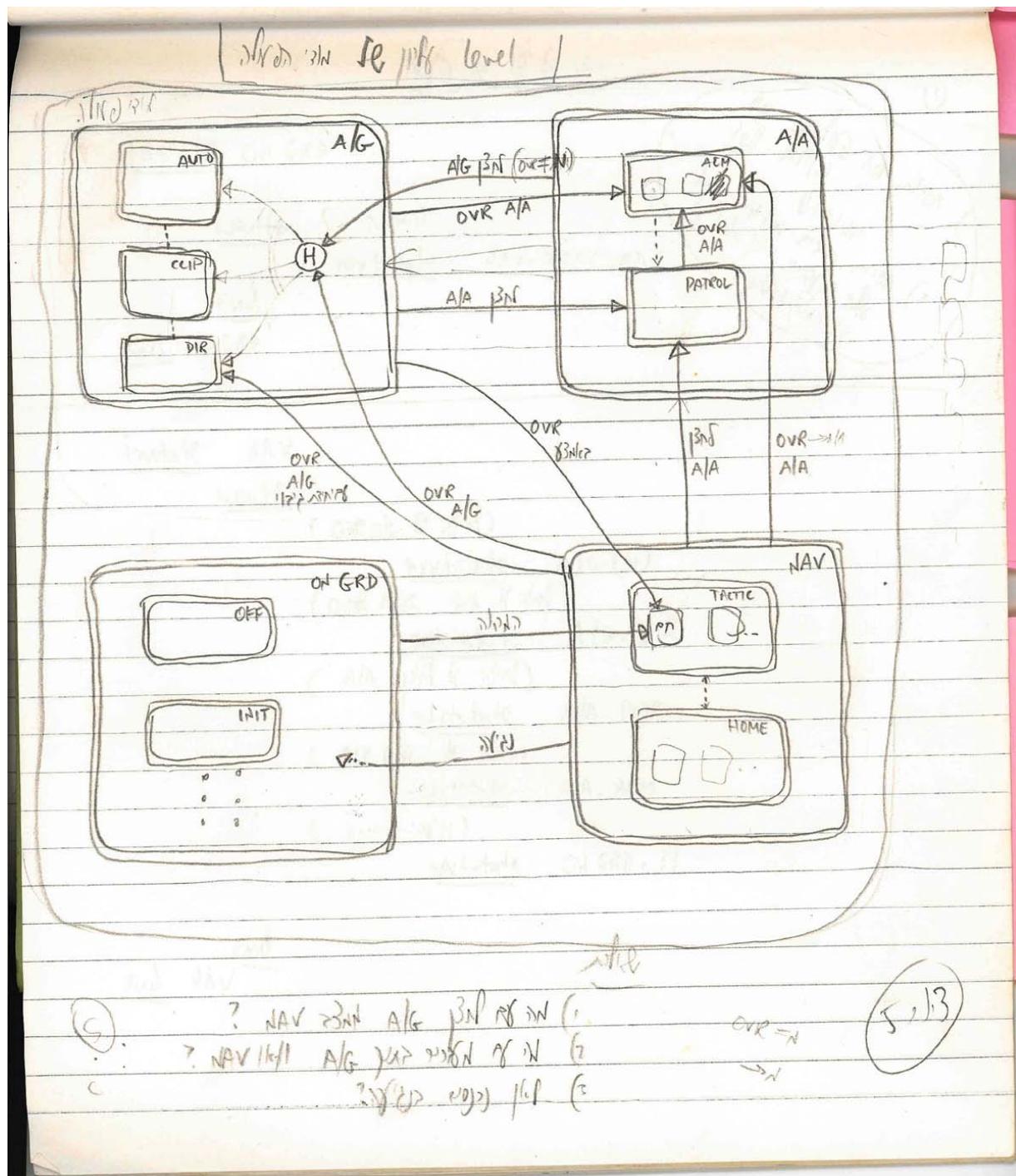


Figure 7. Page from the IAI notes (mid-1983; events in Hebrew) showing a relatively "clean" draft of the top levels of behavior for the main flight modes of the Lavi avionics. These are A/A (air-air), A/G (air-ground), NAV (automatic navigation) and ON GRD (on ground). Note the early use of a *history* connector in the A/G mode.

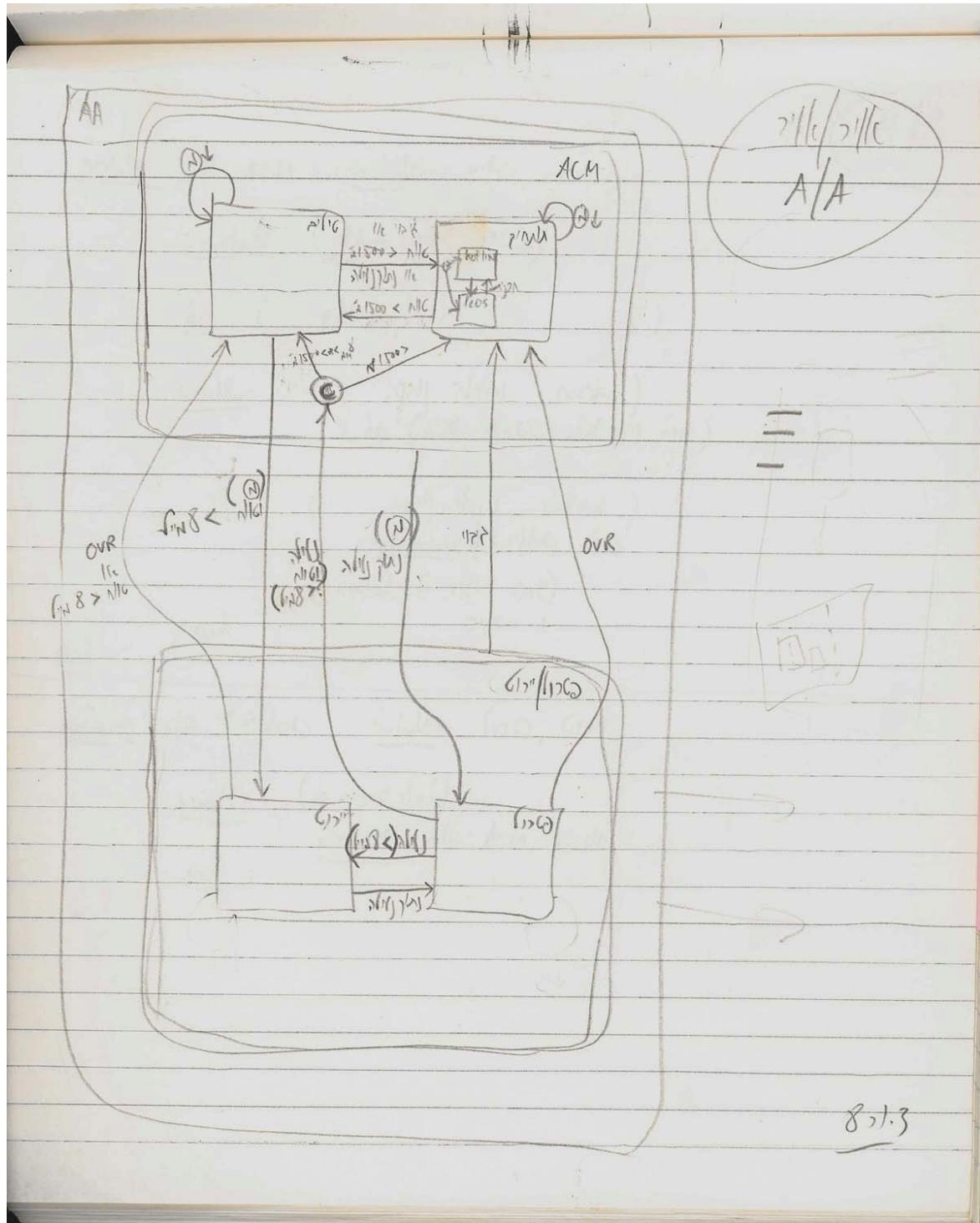


Figure 8. Page from the IAI notes (mid-1983) showing the inner statechart specification of the A/A (air-air) mode for the Lavi avionics.

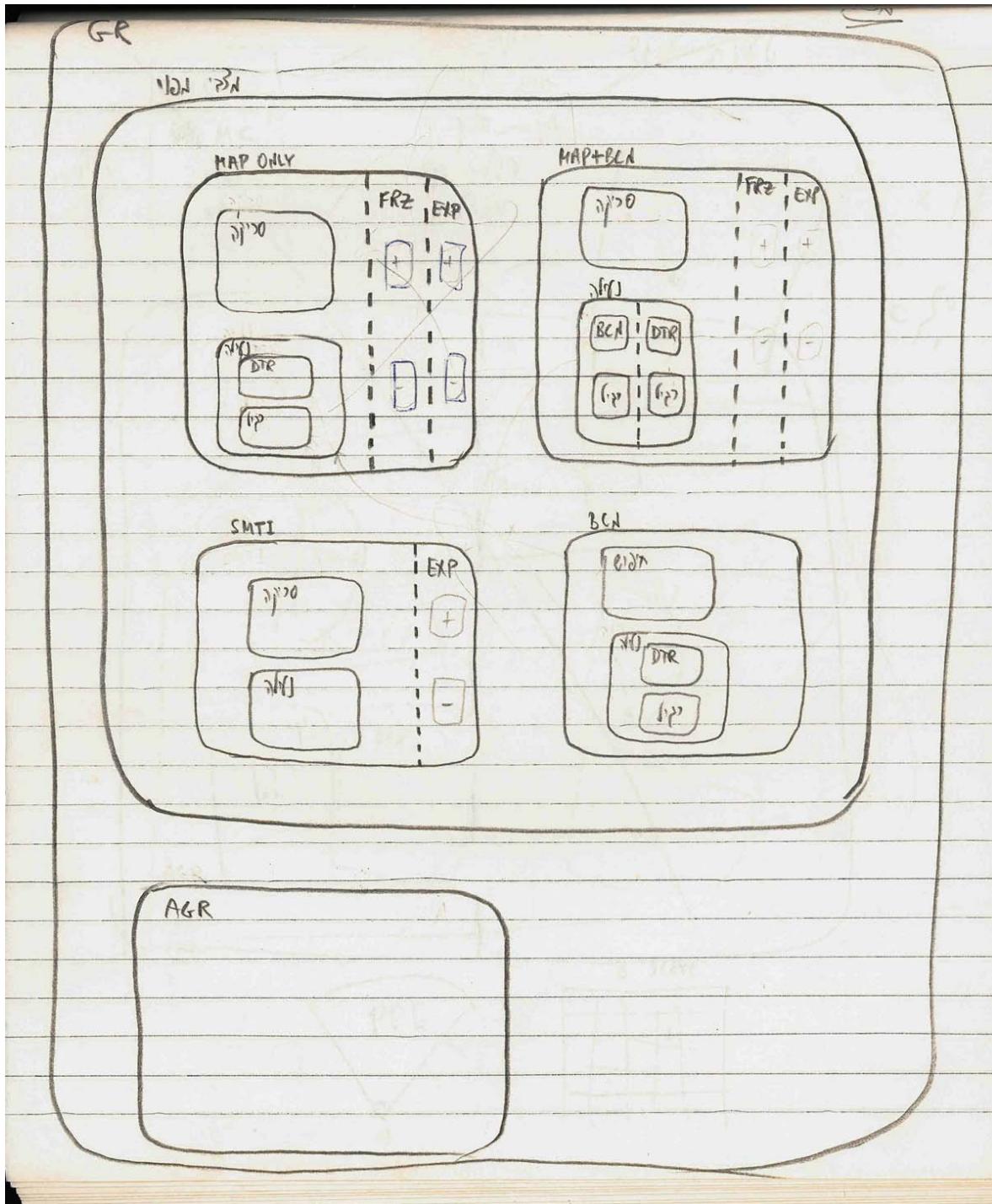


Figure 9. Page from the IAI notes (late 1983) showing multiple-level orthogonality in a complex portion of the Lavi avionics. Most of the orthogonal components on all levels here are not tangible components of the system, but rather exhibit a natural way of conceptually breaking up the state space.

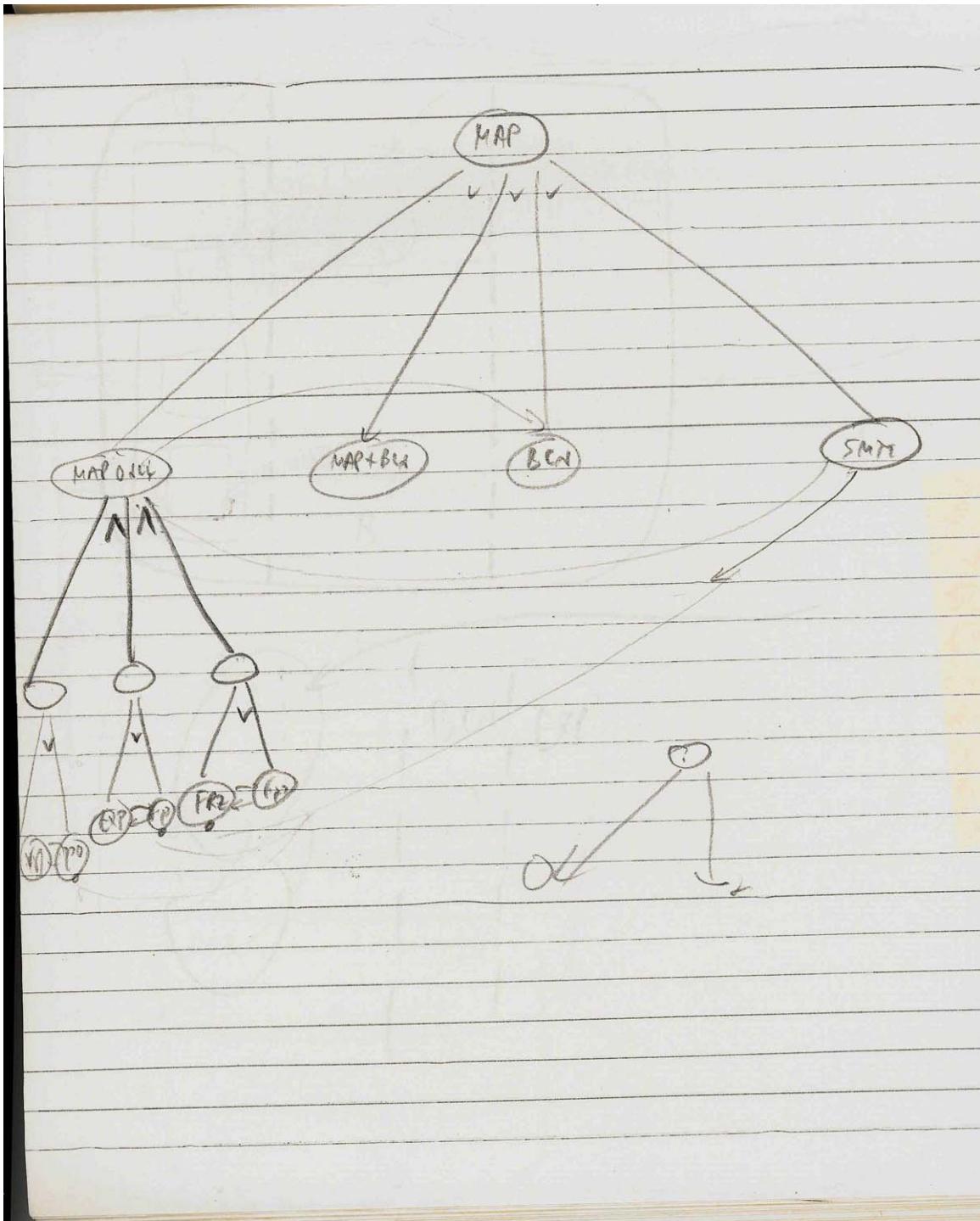


Figure 10. Page from the IAI notes (late 1983) showing an and/or tree rendition of (part of) the state hierarchy in Fig. 9.

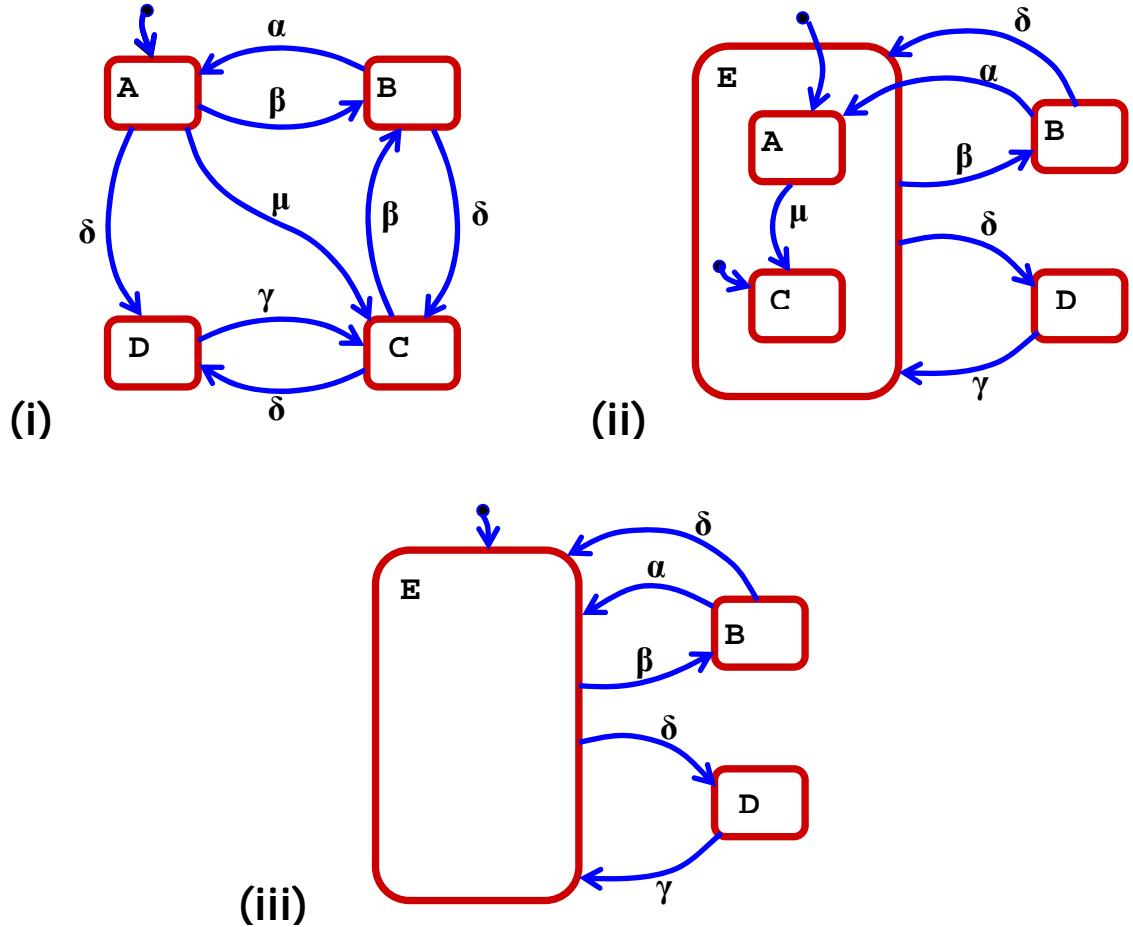
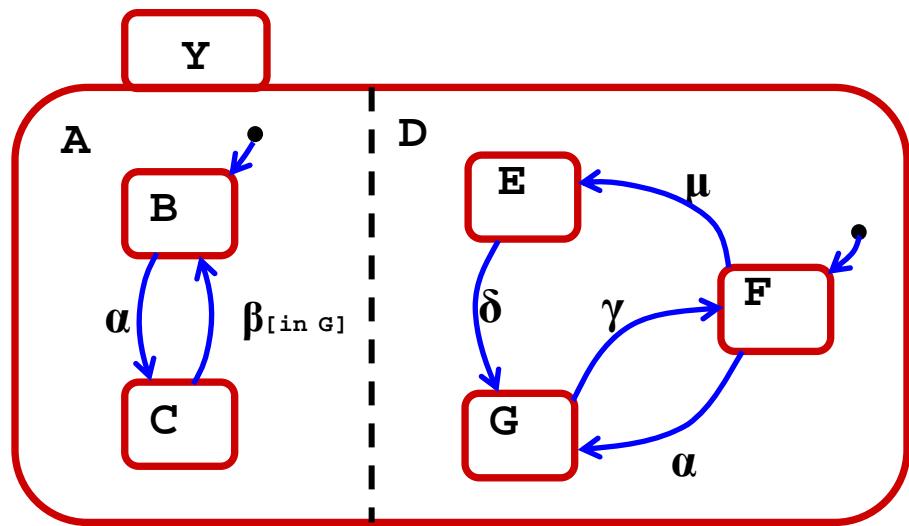
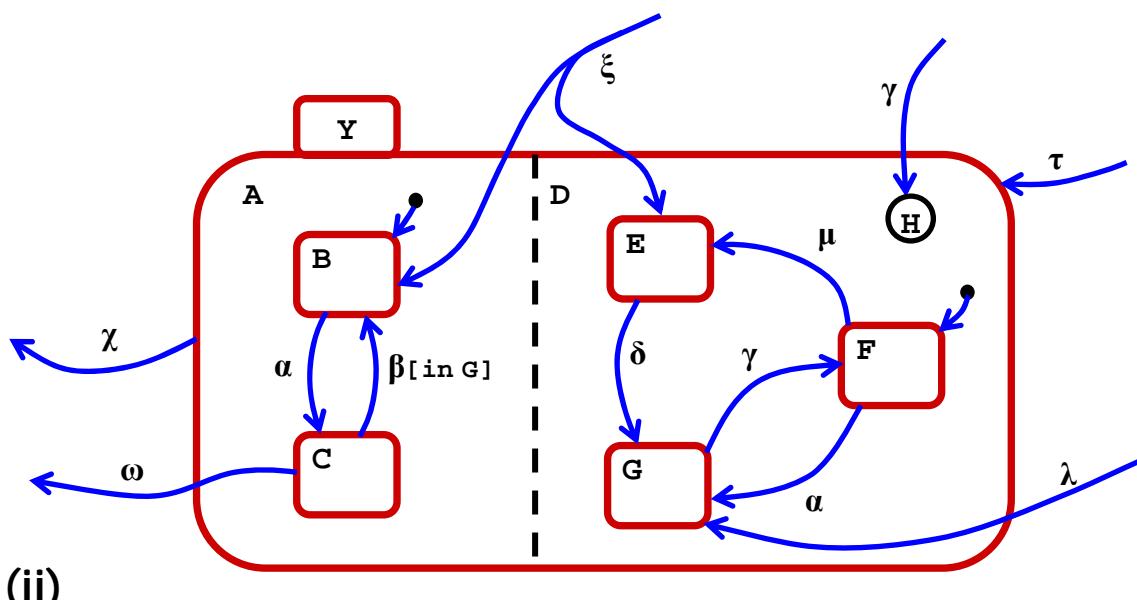


Figure 11. Illustrating hierarchy in statecharts: multi-level states, transitions, default entrances, refinement and abstraction.



(i)



(ii)

Figure 12. Orthogonality in statecharts, with and without out exits from and entrances to other states.

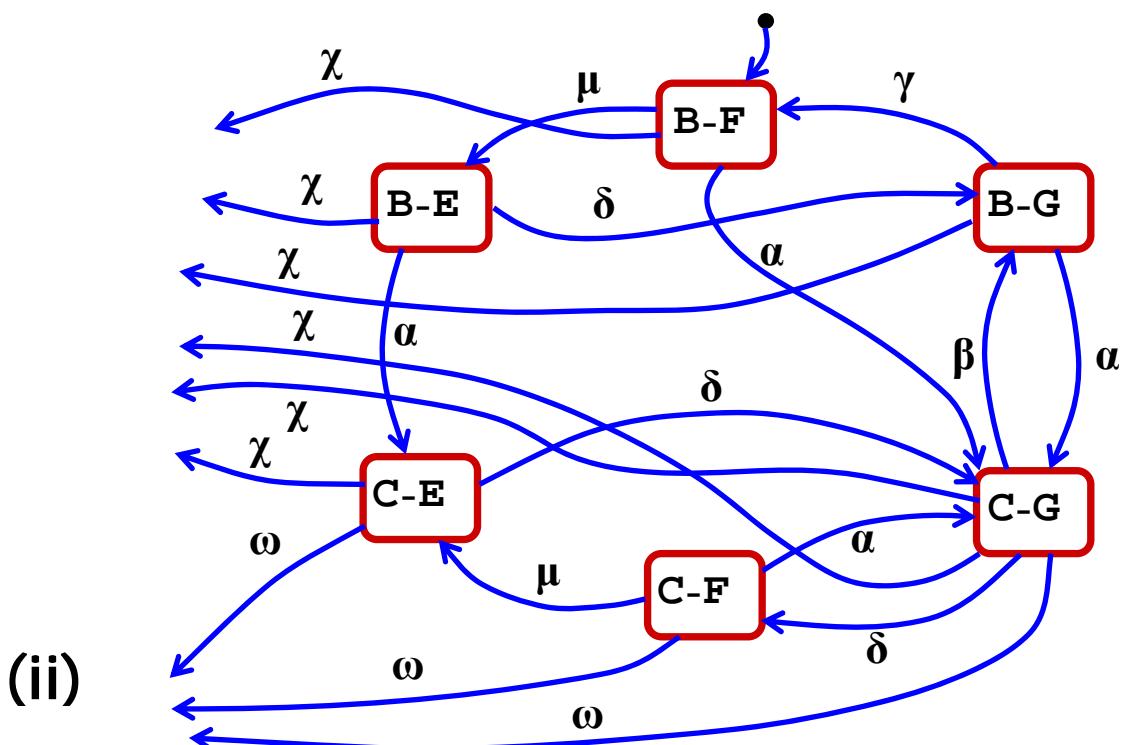
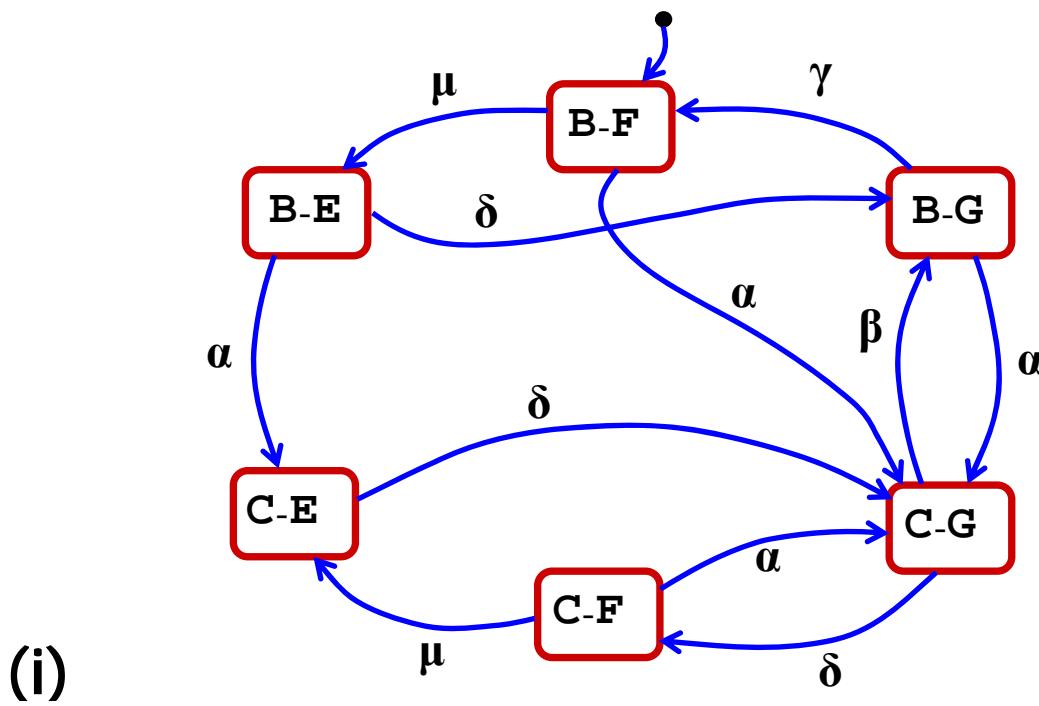


Figure 13. "Flattened" orthogonality-free versions of the two parts of Fig. 12, minus the external entrances in 12(ii). (These are really the Cartesian products.)

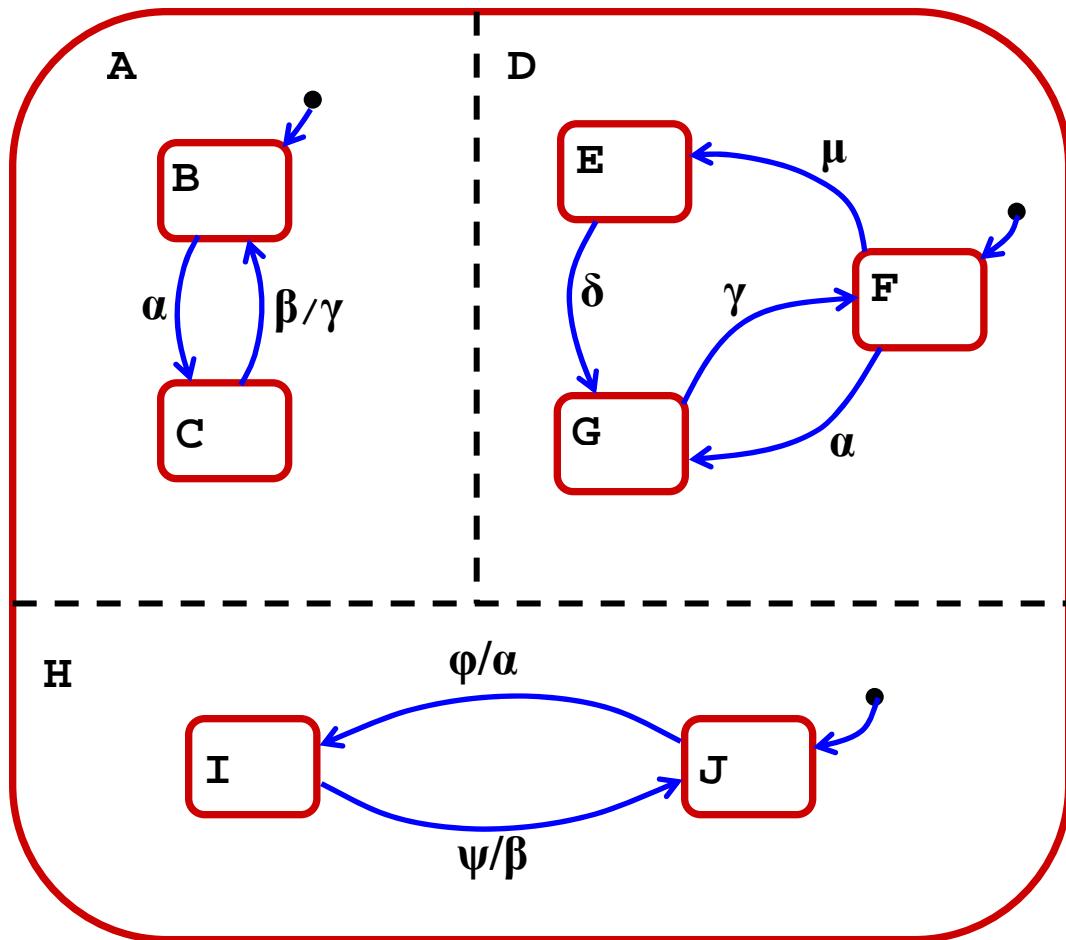
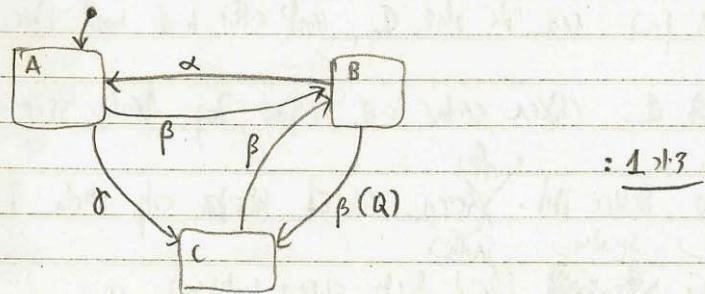


Figure 14. Broadcasting within a single statechart.

(UML) statechart : statecharts

לעתה נסמן מילויים בטבלה;

17 final stage 103 103 103 103 103 103 103



46. B 2nd year 8/10/13, C 2nd year 8/10/13 (

183 A-f oppjor qn. -c-f nysk afn slyf blyf gylf, pjj q 'ygn d.

de, 1973; 3000, 2000, glare of light on the water at the end of the canal

if $f(x) = g(x)$ for all $x \in B$, then $f = g$

A-f very no(p), ~~the~~, ~~the~~

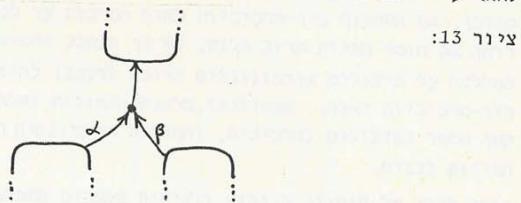
၁၇၈

Figure 15. Page from the IAI notes (late 1983; in Hebrew) showing part of the draft of the internal IAI document reporting on the results of the consulting. Note the first use of the term *statecharts* (top line).

הסמן המשווה של כביסה עצמית, כאילו, ל-H בzeitigר 11 מורה על העבודה שהעדכו משפייע אוניבס בכל תח מבץ של UPDATE אך איינו מתואר כאן עיי מעבר למבץ חדש. כמו כן, הכוונה היא שארוע העדכו מוציא את המערכת מתח מבץ כלשהו ומחזירה מידית שוב ל-UPDATE עיף ההיסטוריה, דהיינו, שוב לאוטו תח מבץ שבו הייתה לפני העדכו. למחרת ציין, ניתן היה לתאר עניין זה עיי ארבע חיצים מהטוג המופיע בzeitigר 12.



נרשא קיזורי דרך גרפים שוביים להקלת על העומס בתרשימים, כגון כתיבת שבוי ארוועים מופרדים עיי. OR על חז אחד במקום שבוי חיצים עם ארוע על כל אחד, או ביקוץ שבוי חיצים או יותר לאחד עיי צומת ביקוץ כמו בzeitigר 13.



האופציה הנוספת שעדין לא תארבו היא של הפיצול האורתוגונלי למרכיבים, המתאים ל-AND. דוגמא (חקלית) שניתנת כאן בzeitigר 14 מתחאת קצתן מן הרמה העלירונה ביותר של מערכת האויבוביקה ללבי, ובבה רואים כמה קוארדיניות אונרולוגיות של מבץ האויבוביקה הכללי.

zeitigר 14

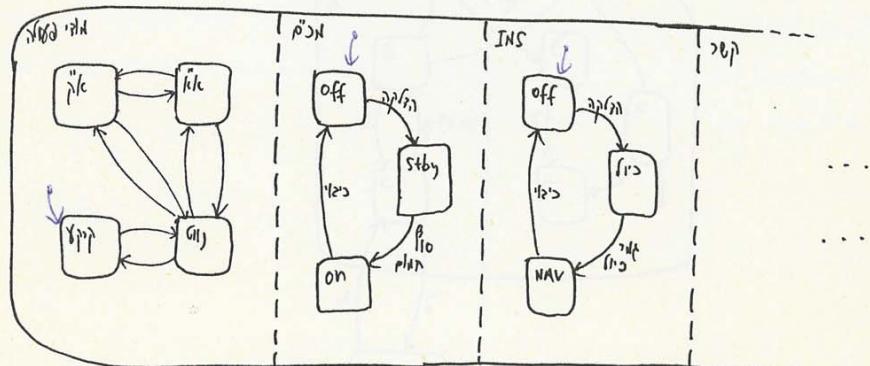


Figure 16. Page 10 of the internal IAI document (December 1983; in Hebrew). The bottom figure shows some of the high-level states of the Lavi avionics, including on the left (in Hebrew...) A/A, A/G, NAV and GRD.

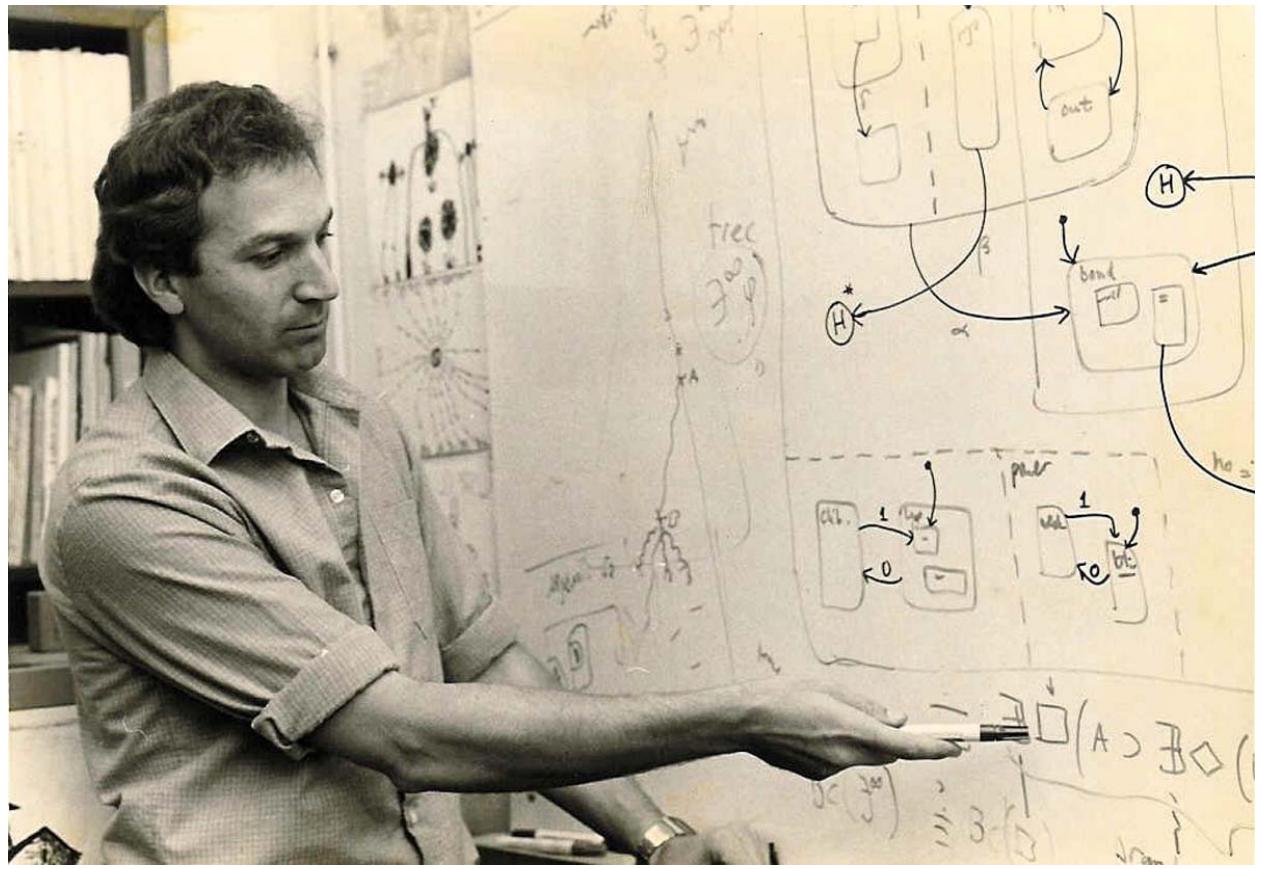


Figure 17. Explaining statecharts (early 1984). Note the temporal logic on the bottom right.

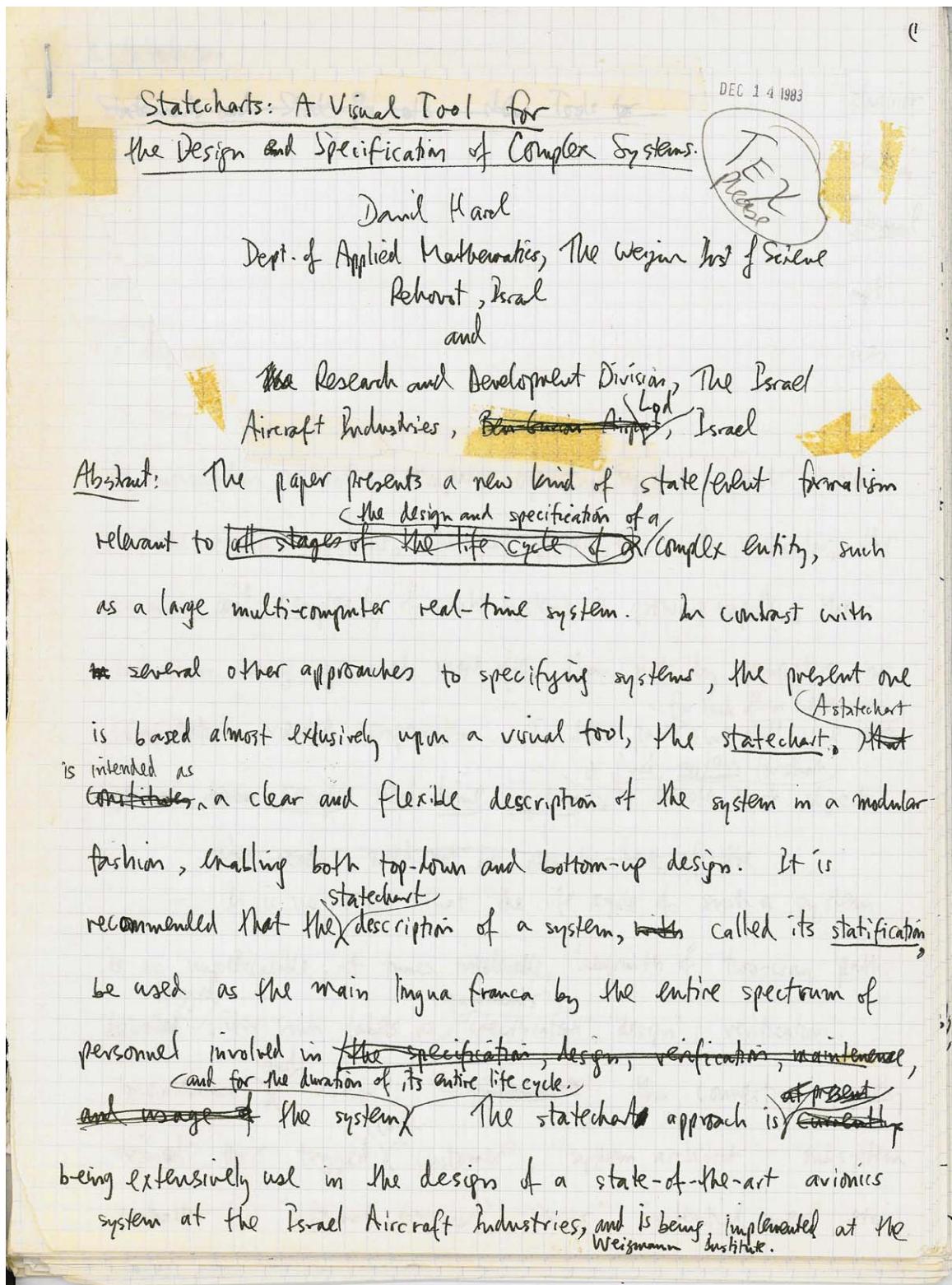


Figure 18. Front page of the first draft of the basic statecharts paper (Dec. 14, 1983). Note the "TeX please" instruction to the typist/secretary; the original title (later changed twice), the use of the word "stratification" for the act of specifying with statecharts (later dropped), and the assertion that the language "is being implemented at the Weizmann Institute" (later changed).

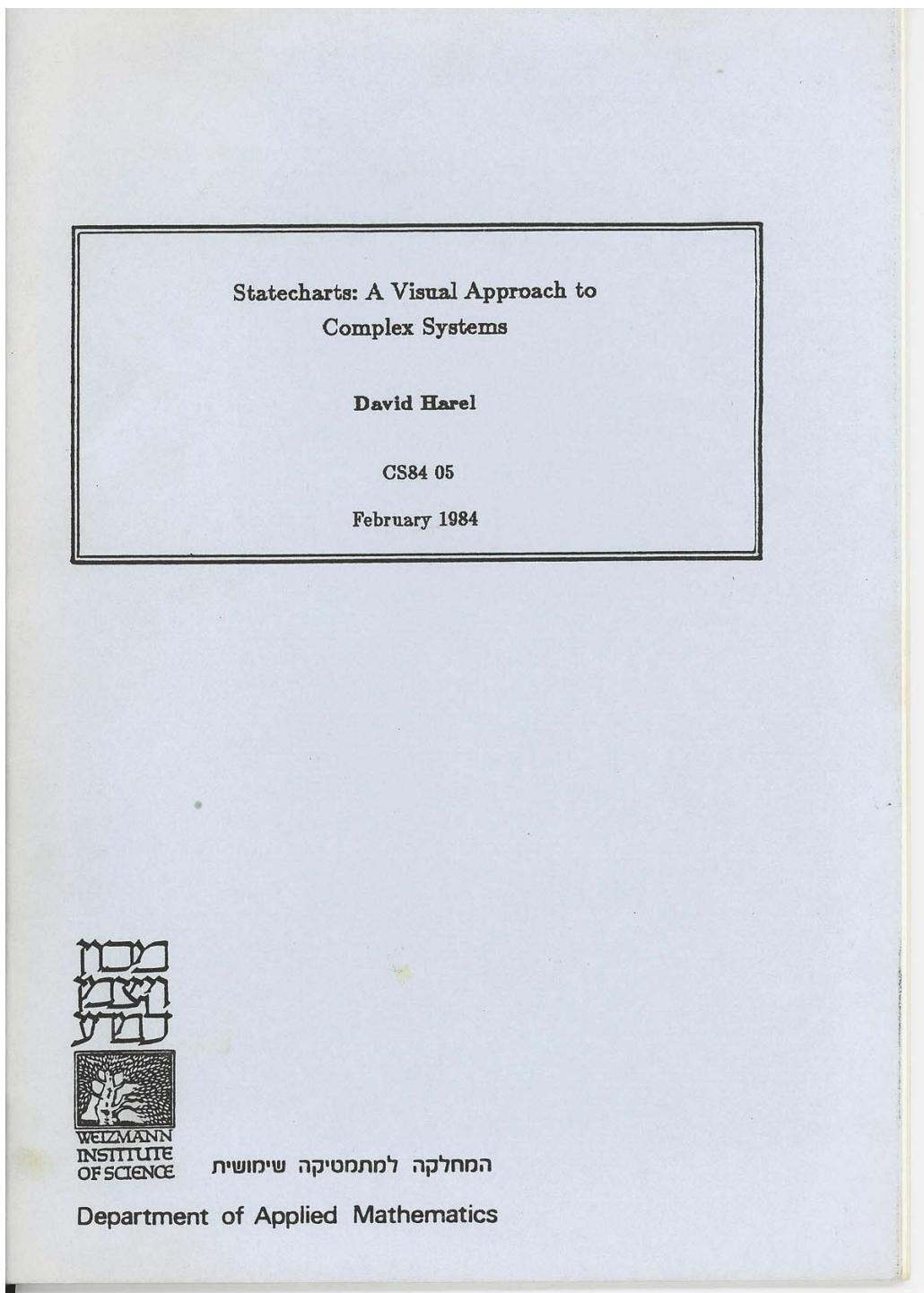


Figure 19. Front page of the technical report version of the basic statecharts paper (February 1984). Note the revised title (later changed again...).

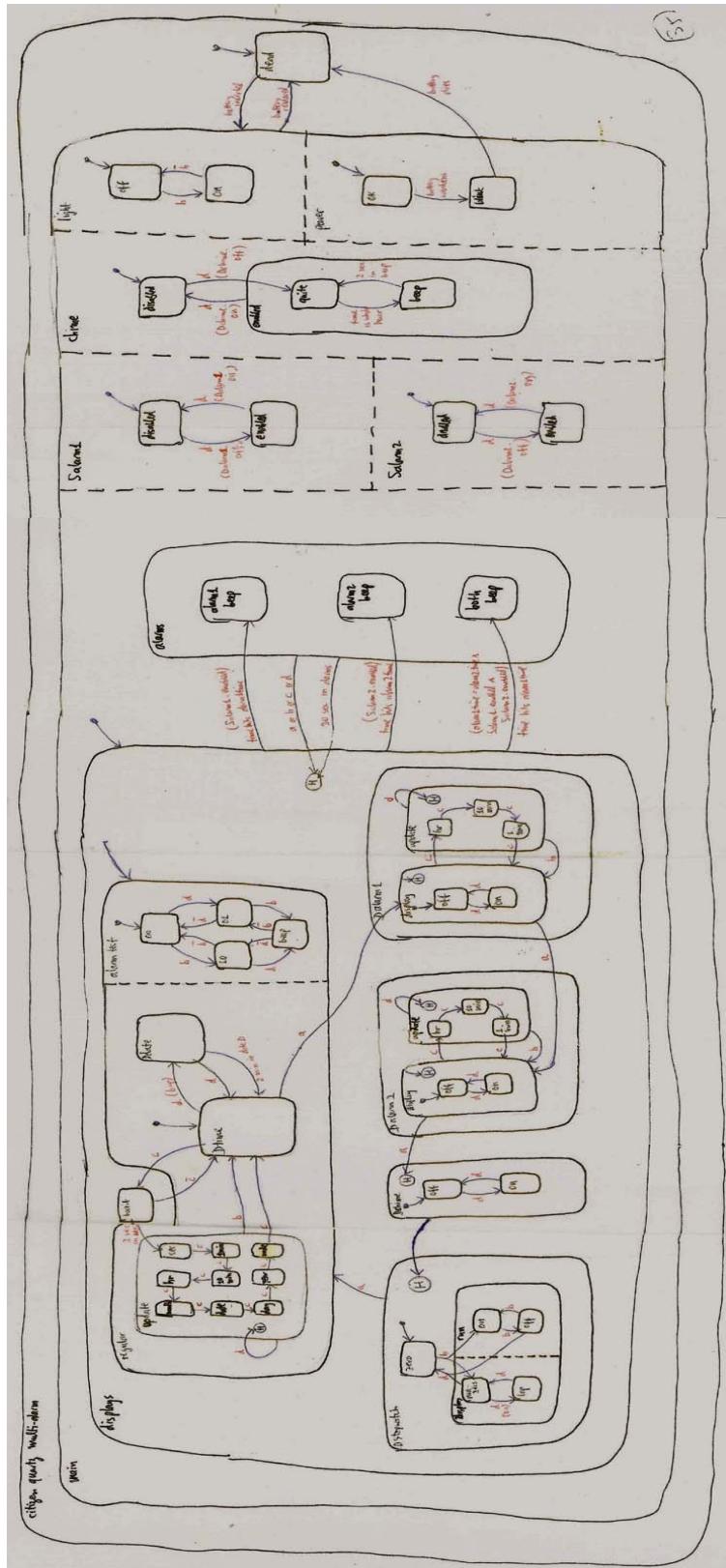


Figure 20. Hand-drawn draft of the Citizen watch statechart (late 1984), as sent to our graphics draftsman for professional drawing.

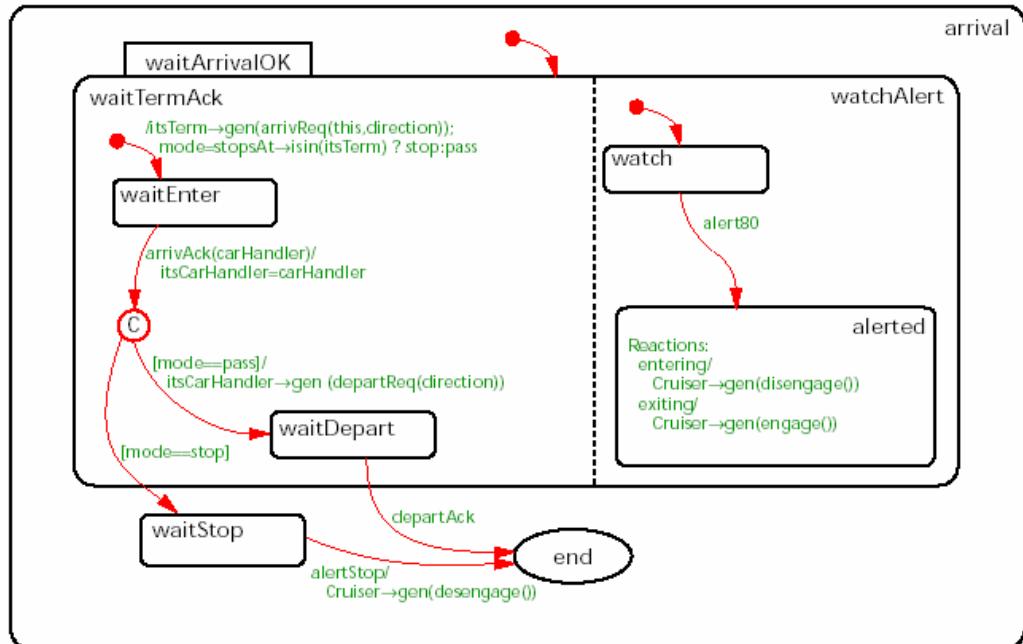
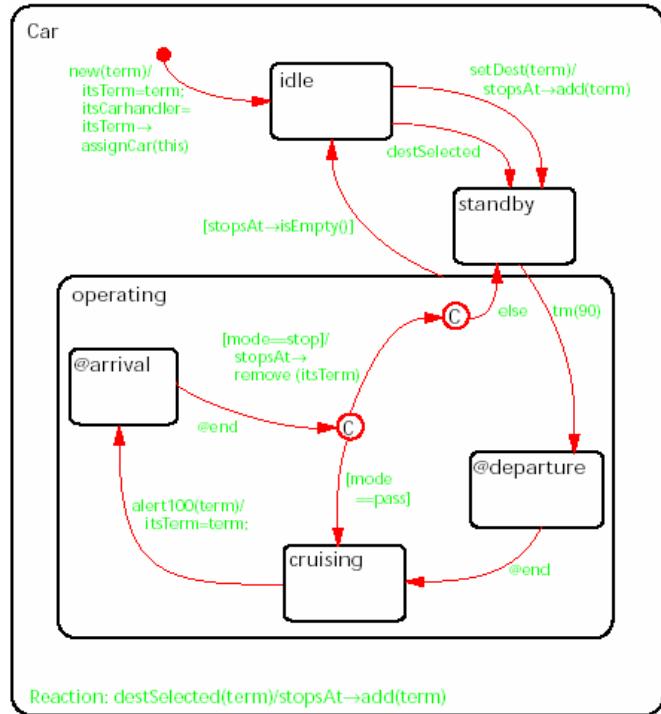


Figure 21. Two object-oriented statecharts for a railcar example, taken from [HG96&97]. Note the C++ action language.

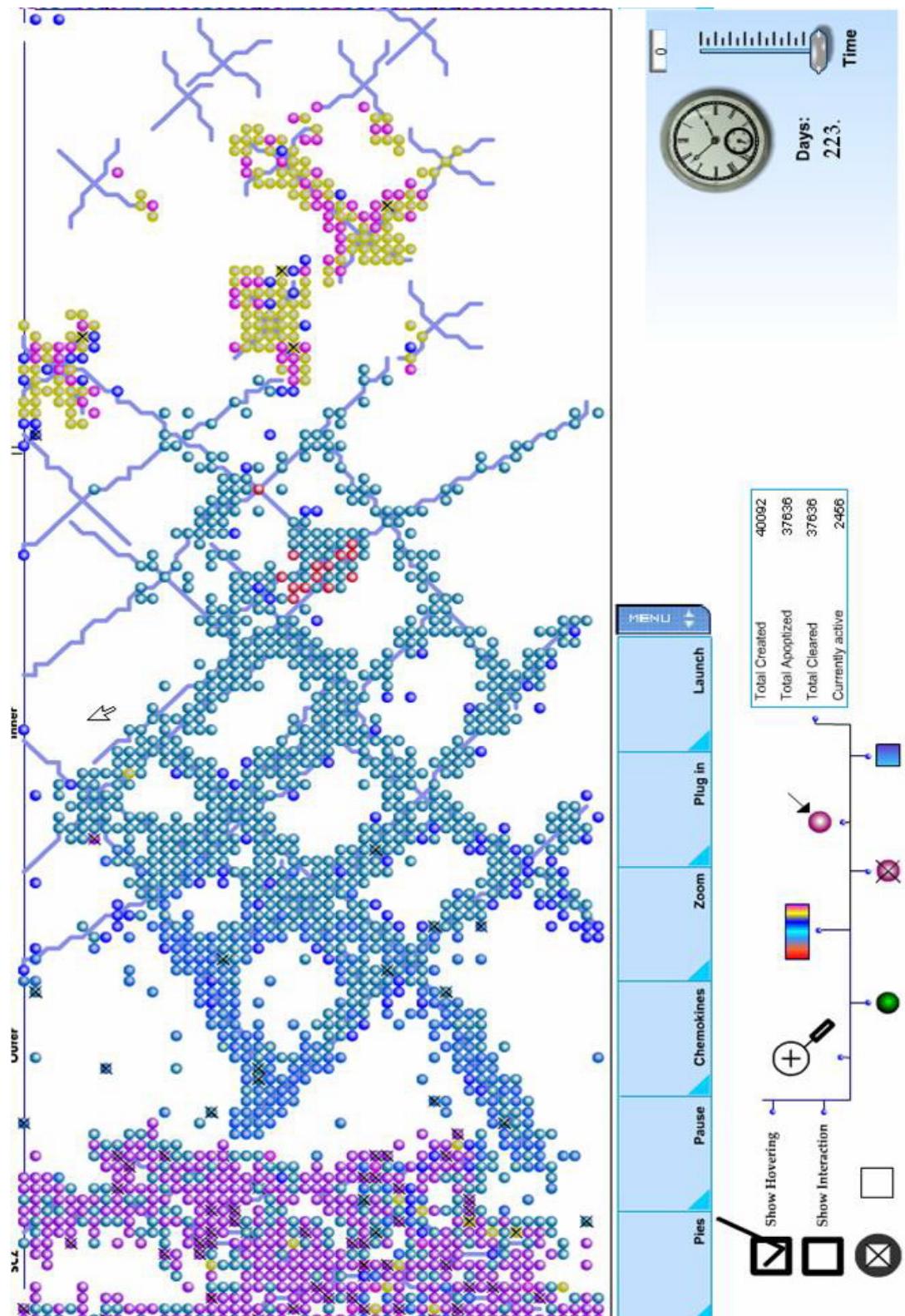


Figure 22. Front end (in Flash) of a model of T cell development in the thymus (from [EHC03]).

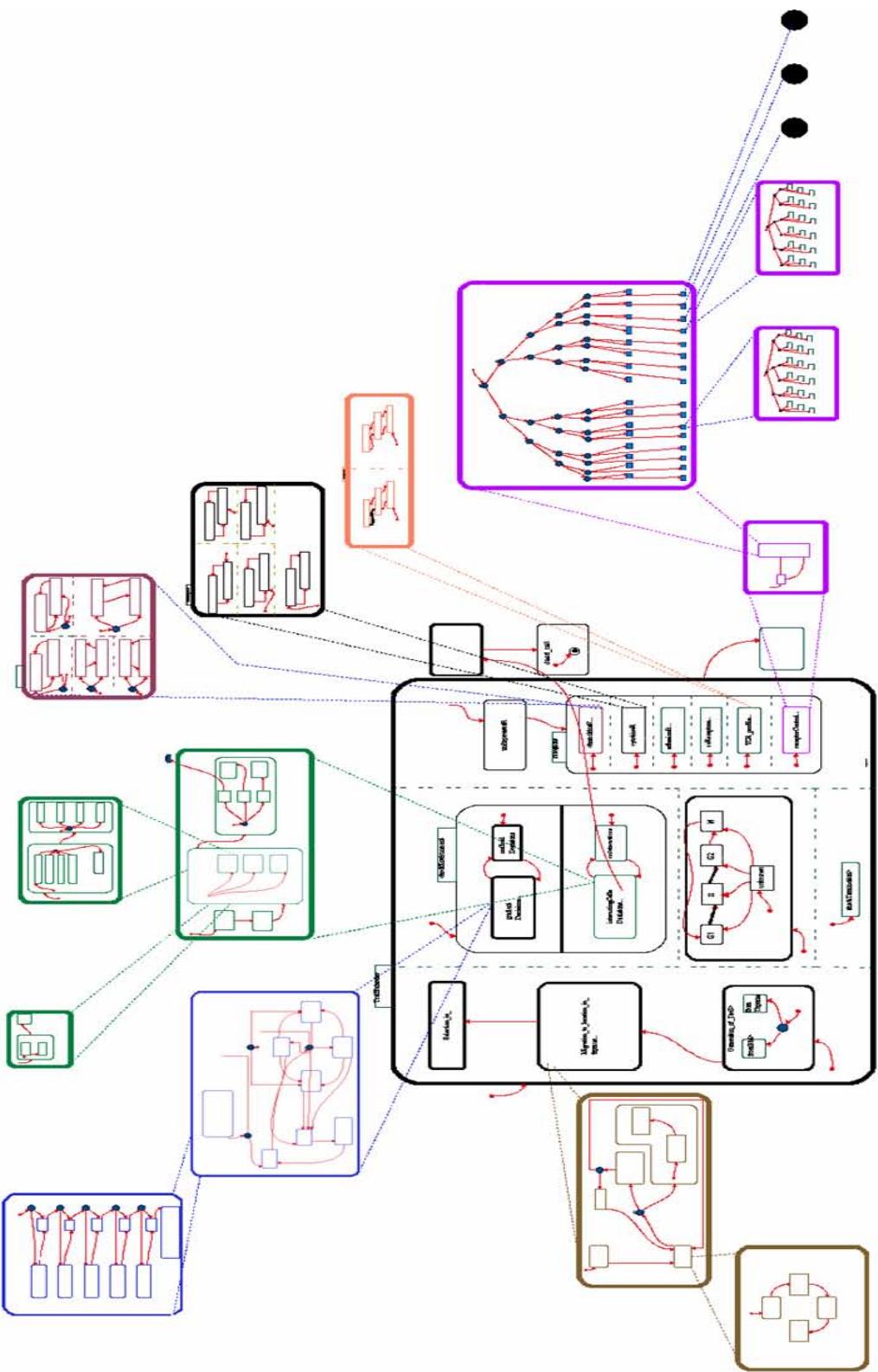


Figure 23. Schematics of parts of the statechart of a single cell from the T cell model shown in Fig. 21 (from [EHC03]).

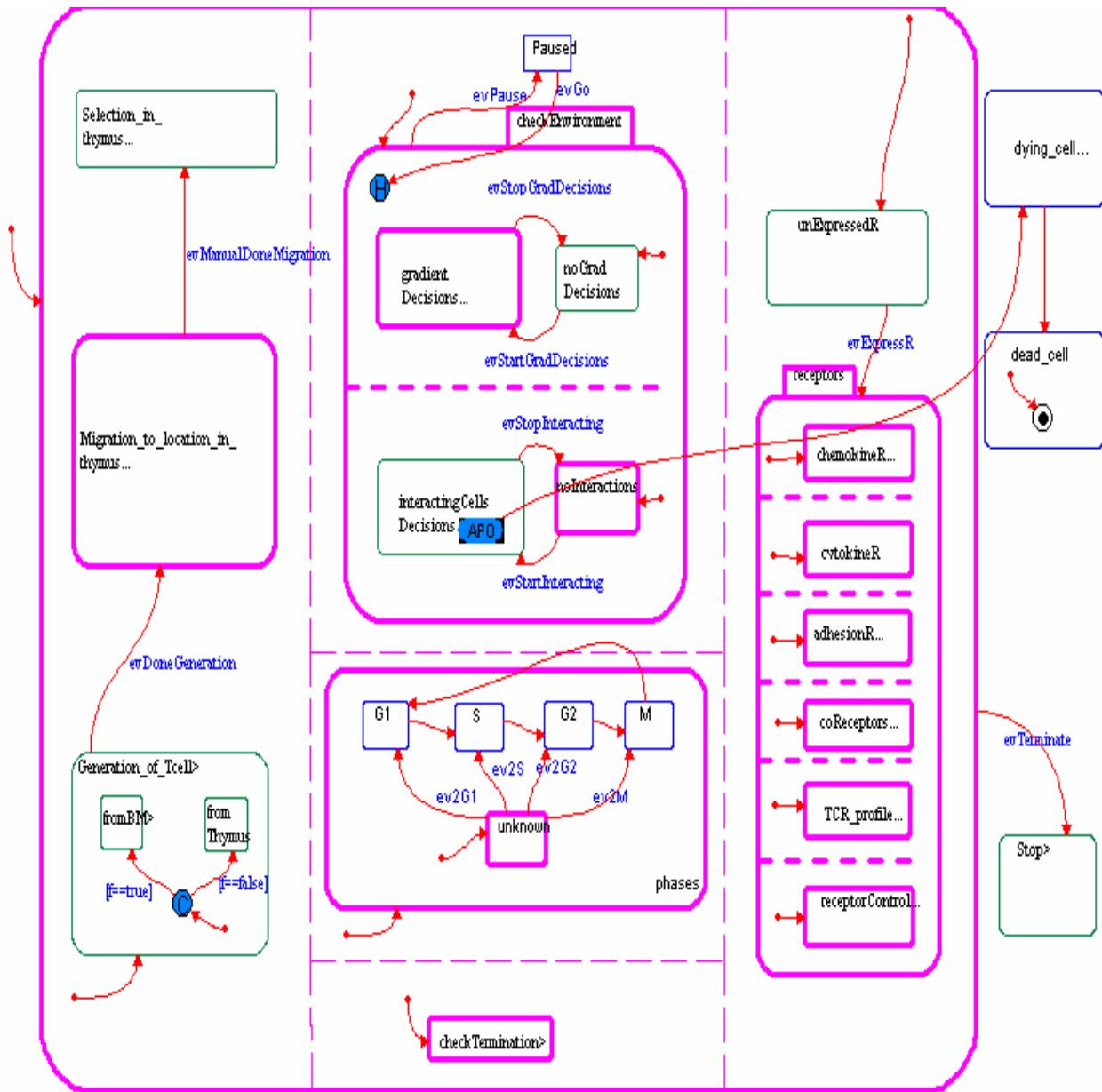


Figure 24. Snapshot of the first few levels of the statechart of a single cell from the T cell model of [EHC03], shown during execution on Rhapsody. The purple states (thick-lined, if you are viewing this in B&W) are the ones the system is in at the moment.

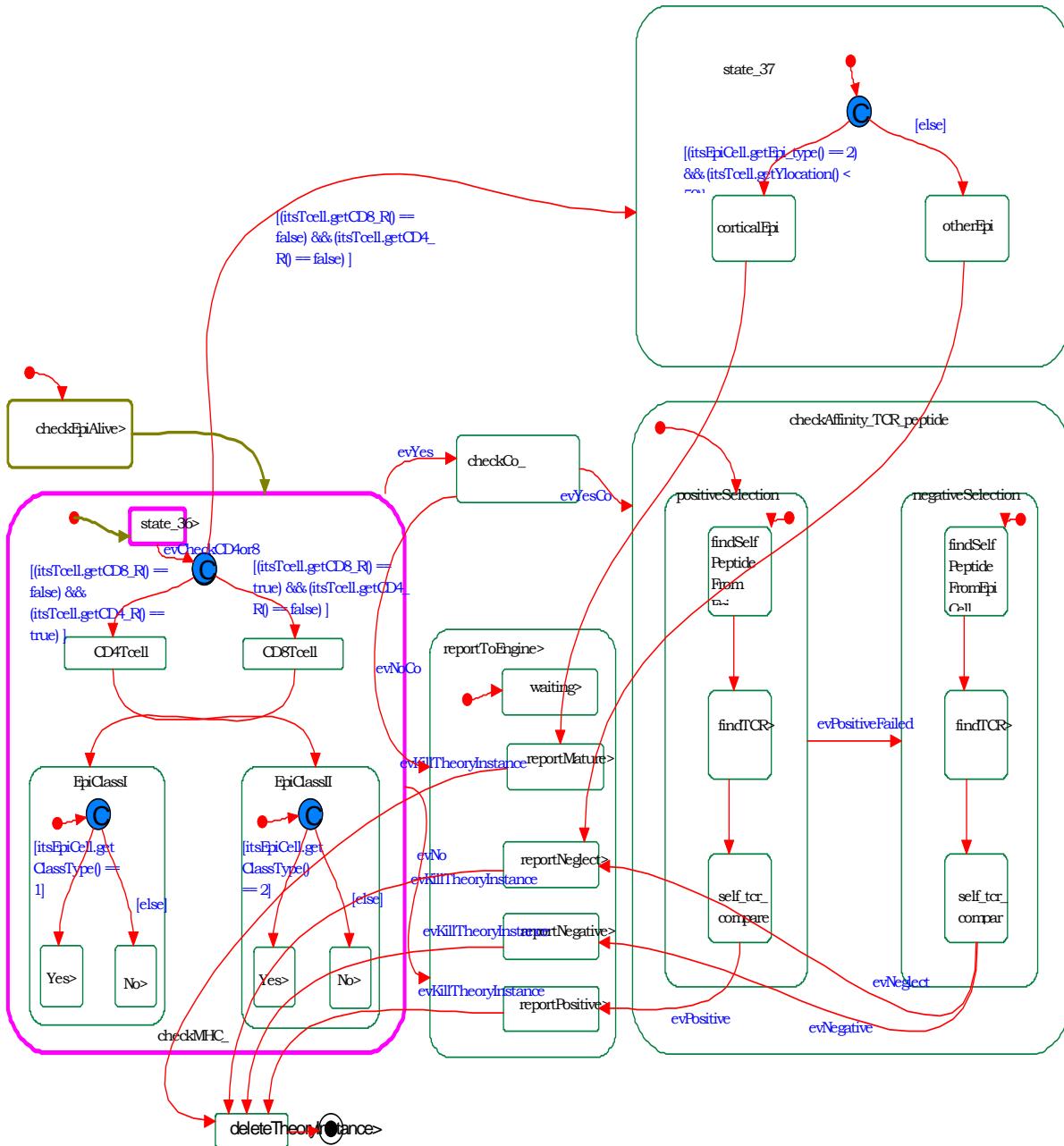


Figure 25. Snapshot of the statechart of the object dealing with the interaction between a potential T cell and an epithelial cell in the model of [EHC03], shown during execution on Rhapsody. The purple states (thick-lined, if you are viewing this in B&W) are the ones the system is in at the moment.

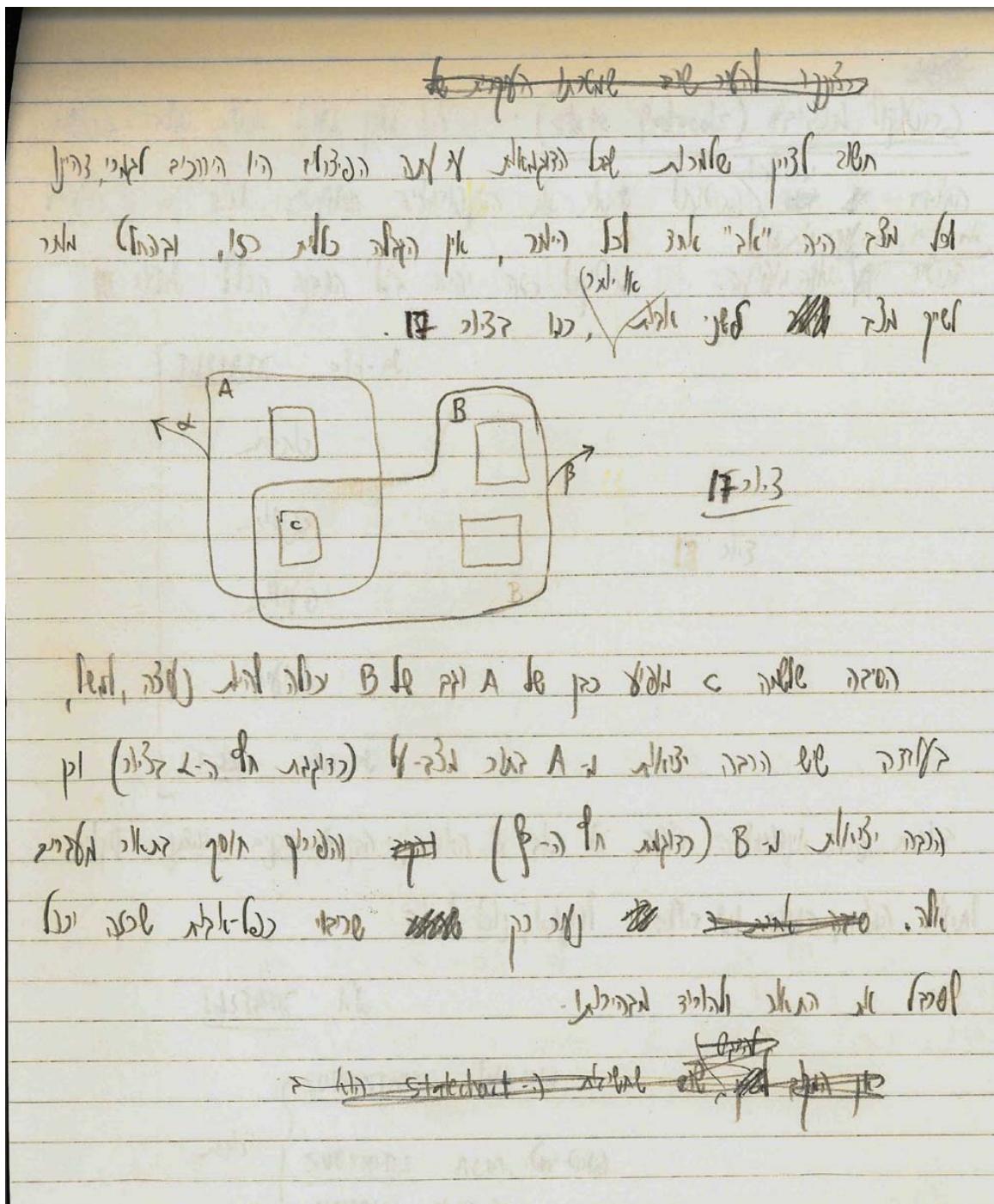


Figure 26. Page from the IAI notebook (late 1983) showing the use of overlapping states, which were never implemented.

References

- [BHM04] D. Barak, D. Harel and R. Marely, "InterPlay: Horizontal Scale-Up and Transition to Design in Scenario-Based Programming", *Lectures on Concurrency and Petri Nets* (J. Desel, W. Reisig and G. Rozenberg, eds.), Lecture Notes in Computer Science, Vol. 3098, Springer-Verlag, 2004, pp. 66–86.
- [B+03] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic and R. de Simone, "The Synchronous Languages Twelve Years Later", *Proc. of the IEEE* **91** (2003), 64–83.
- [BG90] A. Benveniste and P. Le Guernic, "Hybrid Dynamical Systems Theory and the Signal Language", *IEEE Trans. on Automatic Control*, AC-**35** (1990), 535–546.
- [BG92] G. Berry and G. Gonthier, "The Synchronous Programming Language ESTEREL: Design, Semantics, Implementation," *Science of Computer Programming* **19**:2 (1992) 87–152.
- [B94] G. Booch, *Object-Oriented Analysis and Design with Applications*, Benjamin/Cummings, California, 1994.
- [B87] F. P. Brooks, "No Silver Bullet: Essence and Accidents of Software Engineering," *IEEE Computer* **20**:4 (1987) 10–19.
- [CPHP87] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice, "Lustre: A Declarative Language for Programming Synchronous Systems", *Proc. 14th ACM Symp. on Principles of Programming Languages*, ACM Press, 1987, pp. 178–188.
- [CKS81] A.K Chandra, D.C. Kozen and L.J. Stockmeyer, "Alternation", *Journal of the ACM* **28**:1 (1981), 114–133.
- [DH99&01] W. Damm and D. Harel, "LSCs: Breathing Life into Message Sequence Charts", *Formal Methods in System Design* **19**:1 (2001), 45–80. (Preliminary version in *Proc. 3rd IFIP Int. Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS '99)* (P. Ciancarini, A. Fantechi and R. Gorrieri, eds.), Kluwer Academic Publishers, 1999, pp. 293–312.)
- [DH88] D. Drusinsky and D. Harel, "On the Power of Cooperative Concurrency", *Proc. Concurrency '88*, Lecture Notes in Computer Science, Vol. 335, Springer-Verlag, New York, 1988, pp. 74–103.
- [DH94] D. Drusinsky and D. Harel, "On the Power of Bounded Concurrency I: Finite Automata", *J. Assoc. Comput. Mach.* **41** (1994), 517–539.
- [EHC03] S. Efroni, D. Harel and I.R. Cohen, "Towards Rigorous Comprehension of Biological Complexity: Modeling, Execution and Visualization of Thymic T Cell Maturation", *Genome Research* **13** (2003), 2485–2497.
- [EHC05] S. Efroni, D. Harel and I.R. Cohen, "Reactive Animation: Realistic Modeling of Complex Dynamic Systems", *IEEE Computer* **38**:1 (2005), 38–47.
- [EHC07] S. Efroni, D. Harel and I.R. Cohen, "Emergent Dynamics of Thymocyte Development and Lineage Determination", *PLOS Computational Biology* **3**:1 (2007), 127–136.
- [HZ76] M. Hamilton and S. Zeldin, "Higher Order Software: A Methodology for Defining Software", *IEEE Transactions on Software Engineering* **SE-2**:1 (1976), 9–36.
- [H79&80] D. Harel, "And/Or Programs: A New Approach to Structured Programming", *ACM Trans. on Programming Languages and Systems* **2** (1980), 1–17. (Also *Proc. IEEE Specifications for Reliable Software Conf.*, pp. 80–90, Cambridge, Massachusetts, 1979.)
- [H84&87] D. Harel, "Statecharts: A Visual Formalism for Complex Systems", *Science of Computer Programming* **8** (1987), 231–274. (Preliminary version: Technical Report CS84-05, The Weizmann Institute of Science, Rehovot, Israel, February 1984.)
- [H88] D. Harel, "On Visual Formalisms", *Comm. Assoc. Comput. Mach.* **31**:5 (1988), 514–530. (Reprinted in *Diagrammatic Reasoning* (Chandrasekaran et al., eds.), AAAI Press and MIT Press, 1995, pp. 235–271, and in *High Integrity System Specification and Design* (Bowen and Hinchee, eds.), Springer-Verlag, London, 1999, pp. 623–657.)
- [H92] D. Harel, "Biting the Silver Bullet: Toward a Brighter Future for System Development", *IEEE Computer* **25**:1 (1992), 8–20.
- [HG96&97] D. Harel and E. Gery, "Executable Object Modeling with Statecharts", *IEEE Computer* **30**:7 (1997), 31–42. (Also in *Proc. 18th Int. Conf. on Software Engineering*, IEEE Press, 1996, pp. 246–257.)
- [HK92] D. Harel and H.-A. Kahana, "On Statecharts with Overlapping", *ACM Trans. on Software Engineering Method.* **1**:4 (1992), 399–421.
- [HK04] D. Harel and H. Kugler, "The Rhapsody Semantics of Statecharts (or, On the Executable Core of the UML)", *Integration of Software Specification Techniques for Applications in Engineering*, (H. Ehrig et al., eds.), Lecture Notes in Computer Science, Vol. 3147, Springer-Verlag, 2004, pp. 325–354.
- [HKV02] D. Harel, O. Kupferman and M.Y. Vardi, "On the Complexity of Verifying Concurrent Transition Systems", *Information and Computation* **173** (2002), 143–1611.
- [H+88&90] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtul-Trauring and M. Trakhtenbrot, "STATEMATE: A Working Environment for the Development of Complex Reactive Systems", *IEEE Trans. on Software Engineering* **16**:4 (1990), 403–414. (Early version in *Proc. 10th Int. Conf. on Software Engineering*, Singapore, April 1988, pp. 396–406. Reprinted in *Software State-of-the-Art: Selected Papers* (DeMarco and Lister, eds.), Dorset House Publishing, New York, 1990, pp. 322–338, and in *Readings in*

Hardware/Software Co-design (De Michelis, Ernst and Wolf, eds.), Morgan Kaufmann, 2001, pp. 135–146.)

[HM03] D. Harel and R. Marelly, *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*, Springer-Verlag, 2003.

[HN89&96] D. Harel and A. Naamad, "The STATEMATE Semantics of Statecharts", *ACM Trans. on Software Engineering Method.* **5**:4 (1996), 293–333. (Preliminary version appeared as Technical Report, I-Logix, Inc., 1989.)

[HP85] D. Harel and A. Pnueli, "On the Development of Reactive Systems", in *Logics and Models of Concurrent Systems* (K. R. Apt, ed.), NATO ASI Series, Vol. F-13, Springer-Verlag, New York, 1985, pp. 477–498.

[HPSR87] D. Harel, A. Pnueli, J. Schmidt and R. Sherman, "On the Formal Semantics of Statecharts", *Proc. 2nd IEEE Symp. on Logic in Computer Science*, Ithaca, NY, 1987, pp. 54–64.

[HP91] D. Harel and M. Politi, *The Languages of STATEMATE*, Technical Report, I-Logix, Inc., Andover, MA (250 pp.), 1991.

[HP96] D. Harel and M. Politi, *Modeling Reactive Systems with Statecharts: The STATEMATE Approach*, McGraw-Hill, 1998. (This book is no longer in print, but it can be downloaded from my web page.)

[HR04] D. Harel and B. Rumpe, "Meaningful Modeling: What's the Semantics of 'Semantics'?", *IEEE Computer* **37**:10 (2004), 64–72.

[HP87] D. Hatley and I. Pirbhai, *Strategies for Real-Time System Specification*, Dorset House, 1987.

[HKSP78] K.L. Heninger, J.W. Kallander, J.E. Shore and D.L. Parnas, "Software Requirements for the A-7E Aircraft", *NRL Report 3876*, November 1978, 523 pgs.

[HU79] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, 1979.

[HGdR88] C. Huizing, R. Gerth and W.P. de Roever, "Modeling Statecharts Behaviour in a Fully Abstract Way", *Proc. 13th Colloquium on Trees in Algebra and Programming (CAAP '88)*, Lecture Notes in Computer Science, Vol. 299, Springer-Verlag, 2004, pp. 271–294.

[I-L] I-Logix, Inc. Products Web page, <http://www.ilogix.com>.

[KCH01] N. Kam, I.R. Cohen and D. Harel, "The Immune System as a Reactive System: Modeling T Cell Activation with Statecharts", *Bull. Math. Bio.*, to appear. (Extended abstract in *Proc. Visual Languages and Formal Methods* (VLFM'01), part

of *IEEE Symp. on Human-Centric Computing* (HCC'01), 2001, pp. 15–22.)

[LHHR94] N. G. Leveson, M. P. E. Heimdahl, H. Hildreth, and J. D. Reese, "Requirements Specification for Process-Control Systems", *IEEE Transactions on Software Engineering* **20**:9 (1994), 684–707.

[MM85] J. Martin and C. McClure, *Diagramming Techniques for Analysts and Programmers*, Prentice-Hall, 1985.

[MS88] K. Misue and K. Sugiyama, "Compound graphs as abstraction of card systems and their hierarchical drawing," *Inform. Processing Soc.*, Japan, Research Report 88-GC-32-2, 1988, (in Japanese).

[PS91] A. Pnueli and M. Shalev, "What Is in a Step: On the Semantics of Statecharts", *Proc. Symp. on Theoret. Aspects of Computer Software*, Lecture Notes in Computer Science, Vol. 526, Springer-Verlag, 1991, pp. 244–264.

[R85] W. Reisig, *Petri Nets: An Introduction*, Springer-Verlag, 1985.

[RJB99] J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual*, Addison-Wesley, 1999.

[RBPEL91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object Oriented Modeling and Design*, Prentice-Hall, New York, 1991.

[SeCH06] Y. Setty, I.R. Cohen and D. Harel, in preparation, 2006.

[SM91] K. Sugiyama and K. Misue, "Visualization of Structural Information: Automatic Drawing of Compound Digraphs", *IEEE Trans. Systems, Man and Cybernetics* **21**:4 (1991), 876–892.

[SwCH06] N. Swerdliv, I.R. Cohen and D. Harel, "Towards an *in-silico* Lymph Node: A Realistic Approach to Modeling Dynamic Behavior of Lymphocytes", submitted, 2006.

[SGW94] B. Selic, G. Gullekson, and P. Ward, *Real-Time Object-Oriented Modeling*, John Wiley & Sons, New York, 1994.

[UML] Documentation of the Unified Modeling Language (UML), available from the Object Management Group (OMG), <http://www.omg.org>.

[vB94] M. von der Beeck, "A Comparison of Statecharts Variants", *Proc. Formal Techniques in Real Time and Fault Tolerant Systems*, Lecture Notes in Computer Science, Vol. 863, Springer-Verlag, 1994, pp. 128–148.

[WM85] P. Ward and S. Mellor, *Structured Development for Real-Time Systems*, vols. 1–3, Yourdon Press, 1985.

A History of Erlang

Joe Armstrong

Ericsson AB

joe.armstrong@ericsson.com

Abstract

Erlang was designed for writing concurrent programs that “run forever.” Erlang uses concurrent processes to structure the program. These processes have no shared memory and communicate by asynchronous message passing. Erlang processes are lightweight and belong to the language, not the operating system. Erlang has mechanisms to allow programs to change code “on the fly” so that programs can evolve and change as they run. These mechanisms simplify the construction of software for implementing non-stop systems.

This paper describes the history of Erlang. Material for the paper comes from a number of different sources. These include personal recollections, discussions with colleagues, old newspaper articles and scanned copies of Erlang manuals, photos and computer listings and articles posted to Usenet mailing lists.

1. A History of Erlang

1.1 Introduction

Erlang was designed for writing concurrent programs that “run forever.” Erlang uses concurrent processes to structure the program. These processes have no shared memory and communicate by asynchronous message passing. Erlang processes are lightweight and belong to the language and not the operating system. Erlang has mechanisms to allow programs to change code “on the fly” so that programs can evolve and change as they run. These mechanisms simplify the construction of software for implementing non-stop systems.

The initial development of Erlang took place in 1986 at the Ericsson Computer Science Laboratory (the Lab). Erlang was designed with a specific objective in mind: “*to provide a better way of programming telephony applications.*” At the time telephony applications were atypical of the kind of problems that conventional programming languages were designed to solve. Telephony applications are by their nature highly concurrent: a single switch must handle tens or hundreds of thousands of simultaneous transactions. Such transactions are intrinsically distributed and the software is expected to be highly fault-tolerant. When the software that controls telephones fails, newspapers write about it, something which does not happen when a typical desktop application fails. Telephony software must also be changed “on the fly,” that is, without loss of service occurring in the application as code upgrade

Permission to make digital/hard copy of part of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date of appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Permission may be requested from the Publications Dept., ACM, Inc., 2 Penn Plaza, New York, NY 11201-0701, USA, fax:+1(212) 869-0481, permissions@acm.org

©2007 ACM 978-1-59593-766-7/2007/06-ART6 \$5.00

DOI 10.1145/1238844.1238850

<http://doi.acm.org/10.1145/1238844.1238850>

operations occur. Telephony software must also operate in the “soft real-time” domain, with stringent timing requirements for some operations, but with a more relaxed view of timing for other classes of operation.

When Erlang started in 1986, requirements for virtually zero down-time and for in-service upgrade were limited to rather small and obscure problem domains. The rise in popularity of the Internet and the need for non-interrupted availability of services has extended the class of problems that Erlang can solve. For example, building a non-stop web server, with dynamic code upgrade, handling millions of requests per day is very similar to building the software to control a telephone exchange. So similar, that Erlang and its environment provide a very attractive set of tools and libraries for building non-stop interactive distributed services.

From the start, Erlang was designed as a practical tool for getting the job done—this job being to program basic telephony services on a small telephone exchange in the Lab. Programming this exchange drove the development of the language. Often new features were introduced specifically to solve a particular problem that I encountered when programming the exchange. Language features that were not used were removed. This was such a rapid process that many of the additions and removals from the language were never recorded. Appendix A gives some idea of the rate at which changes were made to the language. Today, things are much more difficult to change; even the smallest of changes must be discussed for months and millions of lines of code re-tested after each change is made to the system.

This history is pieced together from a number of different sources. Much of the broad details of the history are well documented in the thesis *Concurrent Functional Programming for Telecommunications: A Case Study for Technology Introduction* [12], written by the head of the Computer Science Lab, Bjarne Däcker. This thesis describes the developments at the lab from 1984 to 2000, and I have taken several lengthy quotes from the thesis. In 1994, Bjarne also wrote a more light-hearted paper [11] to celebrate the tenth anniversary of the lab. Both these papers have much useful information on dates, times and places that otherwise would have been forgotten.

Many of the original documents describing Erlang were lost years ago, but fortunately some have survived and parts of them are reproduced here. Many things we take for granted today were not self-evident to us twenty years ago, so I have tried to expose the flow of ideas in the order they occurred and from the perspective of the time at which they occurred. For comparison, I will also give a modern interpretation or explanation of the language feature or concept involved.

This history spans a twenty-year period during which a large number of people have contributed to the Erlang system. I have done my best to record accurately who did what and when. This is not an easy task since often the people concerned didn’t write any comments in their programs or otherwise record what they were doing, so I hope I haven’t missed anybody out.

2. Part I : Pre-1985. Conception

2.1 Setting the scene

Erlang started in the Computer Science Laboratory at Ericsson Telecom AB in 1986. In 1988, the Lab moved to a different company called Ellemtel in a south-west suburb of Stockholm. Ellemtel was jointly owned by LM Ericsson and Televerket, the Swedish PTT, and was the primary centre for the development of new switching systems.

Ellemtel was also the place where Ericsson and Televerket had jointly developed the AXE telephone exchange. The AXE, which was first produced in 1974 and programmed in PLEX, was a second-generation SPC¹ exchange which by the mid-1980s generated a large proportion of Ericsson's profits.

Developing a new telephone exchange is a complex process which involves the efforts of hundreds, even thousands of engineers. Ellemtel was a huge melting pot of ideas that was supposed to come up with new brilliant products that would repeat the success of the AXE and generate profits for Ericsson and Televerket in the years to come.

Initially, when the computer science lab moved to Ellemtel in 1988 it was to become part of an exciting new project to make a new switch called AXE-N. For various reasons, this plan never really worked out, so the Lab worked on a number of projects that were not directly related to AXE-N. Throughout our time at Ellemtel, there was a feeling of competition between the various projects as they vied with each other on different technical solutions to what was essentially the same problem. The competition between the two groups almost certainly stimulated the development of Erlang, but it also led to a number of technical "wars"—and the consequences of these wars was probably to slow the acceptance of Erlang in other parts of the company.

The earliest motivation for Erlang was "*to make something like PLEX, to run on ordinary hardware, only better.*"

Erlang was heavily influenced by PLEX and the AXE design. The AXE in turn was influenced by the earlier AKE exchange. PLEX is a programming language developed by Göran Hemdahl at Ericsson that was used to program AXE exchanges. The PLEX language was intimately related to the AXE hardware, and cannot be sensibly used for applications that do not run on AXE exchanges. PLEX had a number of language features that corrected shortcomings in the earlier AKE exchange.

In particular, some of the properties of the AXE/PLEX system were viewed as mandatory. Firstly, it should be possible to change code *"on the fly,"* in other words, code change operations should be possible without stopping the system. The AKE was plagued with "pointer" problems. The AKE system manipulated large numbers of telephone calls in parallel. The memory requirements for each call were variable and memory was allocated using linked lists and pointer manipulation. This led to many errors. The design of the AXE and PLEX used a mixture of hardware protection and data copying that eliminated the use of pointers and corrected many of the errors in the AKE. This in its turn was the inspiration of the process and garbage-collected memory strategy used in Erlang.

At this stage it might be helpful to describe some of the characteristics of the concurrency problems encountered in programming a modern switching system. A switching system is made from a number of individual switches. Individual switches typically handle tens to hundreds of thousands of simultaneous calls. The switching system must be capable of handling millions of calls and must tolerate the failure of individual switches, providing uninterrupted services to the user. Usually the hardware and software is divided

into two planes called the control and media planes. The control plane is responsible for controlling the calls, the media plane is responsible for data transmission. When we talk in loose terms about "telephony software" we mean the control-plane software.

Typically, the software for call control is modeled using finite state machines that undergo state transitions in response to protocol messages. From the software point of view, the system behaves as a very large collection of parallel processes. At any point in time, most of the processes are waiting for an event caused by the reception of a message or the triggering of a timer. When an event occurs, the process does a small amount of computation, changes state, possibly sends messages to other processes and then waits for the next event. The amount of computation involved is very small.

Any switching system must therefore handle hundreds of thousands of very lightweight processes where each process performs very little computation. In addition, software errors in one process should not be able to crash the system or damage any other processes in the system. One problem to be solved in any system having very large numbers of processes is how to protect the processes from memory corruption problems. In a language with pointers, processes are protected from each other using memory-management hardware and the granularity of the page tables sets a lower limit to the memory size of a process. Erlang has no pointers and uses a garbage collectible memory, which means that it is impossible for any process to corrupt the memory of another process. It also means that the memory requirements for an individual process can be very small and that all memory for all processes can be in the same address space without needing memory protection hardware.

It is important to note that in the AXE/PLEX system and in Erlang, explicit processes are part of the programming language itself and not part of the underlying operating system. There is a sense in which both Erlang and PLEX do not need most of the services of the underlying operating system since the language itself provides both memory management and protection between parallel processes. Other operating system services, like resource allocation and device drivers needed to access the hardware, can easily be written in C and dynamically linked into the Erlang runtime system.

At the time Erlang was first implemented, the view that processes were part of the language rather than the operating system was not widely held (even today, it is a minority view). The only languages having this view of the world that we were aware of at that time were Ada (with tasks), EriPascal (an Ericsson dialect of Pascal with concurrent processes), Chill (with processes), PLEX (with its own special form of processes implemented in hardware) and Euclid. When I first started working at Ericsson, I was introduced to the Ericsson software culture by Mike Williams, who also worked in the Lab. Mike had worked with concurrent systems for many years, mostly in PL163, and it was he who hammered into my brain the notion that three properties of a programming language were central to the efficient operation of a concurrent language or operating system. These were: 1) the time to create a process. 2) the time to perform a context switch between two different processes and 3) the time to copy a message between two processes. The performance of any highly-concurrent system is dominated by these three times.

The final key idea inherited from the AXE/PLEX culture was that the failure of a process or of hardware should only influence the immediate transaction involved and that all other operations in the machine should progress as if no failures had occurred. An immediate consequence of this on the Erlang design was to forbid dangling pointers between different processes. Message passing had to be implemented by copying message buffers between the memory spaces of the different processes involved and not by passing point-

¹ Stored program control—an ancient telecoms term meaning "computer controlled."

1	Handling a very large number of concurrent activities
2	Actions to be performed at a certain point of time or within a certain time
3	Systems distributed over several computers
4	Interaction with hardware
5	Very large software systems
6	Complex functionality such as feature interaction
7	Continuous operation over several years
8	Software maintenance (reconfiguration, etc.) without stopping the system
9	Stringent quality and reliability requirements
10	Fault tolerance both to hardware failures and software errors

Table 1. Requirements of a programming language for telecommunication switching systems (from [12]).

ers to a common memory pool. At an early stage we rejected any ideas of sharing resources between processes because of the difficulties of error handling. In many circumstances, error recovery is impossible if part of the data needed to perform the error recovery is located on a remote machine and if that remote machine has crashed. To avoid this situation and to simplify the process, we decided that all processes must always have enough local information to carry on running if something fails in another part of the system. Programming with mutexes and shared resources was just too difficult to get right in a distributed system when errors occurred.

In cases where consistency is required in distributed systems, we do not encourage the programmer to use the low-level Erlang language primitives but rather the library modules written in Erlang. The Erlang libraries provide components for building distributed systems. Such components include mnesia, which is a distributed real-time transaction database, and various libraries for things like leadership election and transaction memories. Our approach was always not to hard-wire mechanisms into the language, but rather to provide language primitives with which one could construct the desired mechanisms. An example is the remote procedure call. There is no remote procedure call primitive in Erlang, but a remote procedure call can be easily made from the Erlang send and receive primitives.

We rapidly adopted a philosophy of message passing by copying and no sharing of data resources. Again, this was counter to the mood of the time, where threads and shared resources protected by semaphores were the dominant way of programming concurrent systems. Robert Virding and I always argued strongly against this method of programming. I clearly remember attending several conferences on distributed programming techniques where Robert and I would take turns at asking the question “What happens if one of the nodes crashes?” The usual answer was that “the system won’t work” or “our model assumes that there are no failures.” Since we were interested in building highly reliable distributed systems that should never stop, these were not very good answers.

In order to make systems reliable, we have to accept the extra cost of copying data between processes and always making sure that the processes have enough data to continue by themselves if other processes crash.

2.2 Requirements, requirements, requirements ...

The AXE/PLEX heritage provided a set of requirements that any new programming language for programming telecommunications applications must have, as shown in Table 1.

These requirements were pretty typical. Existing systems solved these problems in a number of ways, sometimes in the programming language, sometimes in the operating systems and sometimes

in application libraries. The goal of the Lab was to find better ways of programming telecoms systems subject to such requirements.

Our method of solving these problems was to program POTS² over and over again in a large number of different programming languages and compare the results. This was done in a project called SPOTS, (SPOTS stood for SPC for POTS). Later the project changed name to DOTS (Distributed SPOTS) and then to LOTS, “Lots of DOTS.” The results of the SPOTS project were published in [10].

2.3 SPOTS, DOTS, LOTS

In 1985, when I joined the Lab, SPOTS had finished and DOTS was starting. I asked my boss Bjarne Däcker what I should do. He just said “solve Ericsson’s software problem.” This seemed to me at the time a quite reasonable request—though I now realise that it was far more difficult than I had imagined. My first job was to join the ongoing DOTS experiment.

Our lab was fortunate in possessing a telephone exchange (an Ericsson MD110 PABX) that Per Hedeland had modified so that it could be controlled from a VAX11/750. We were also lucky in being the first group of people in the company to get our hands on a UNIX operating system, which we ran on the VAX. What we were supposed to do was “to find better ways of programming telephony” (a laudable aim for the members of the computer science lab of a telecommunications company). This we interpreted rather liberally as “program basic telephony in every language that will run on our Unix system and compare the results.” This gave us ample opportunities to a) learn new programming languages, b) play with Unix and c) make the phones ring.

In our experiments, we programmed POTS in a number of different languages, the only requirement being that the language had to run on 4.2 BSD UNIX on the Vax 11/750. The languages tried in the SPOTS project were Ada, Concurrent Euclid, PFL, LPL0, Frames and CLU.

Much of what took place in the POTS project set the scene for what would become Erlang, so it is interesting to recall some of the conclusions from the SPOTS paper. This paper did not come down heavily in favour of any particular style of programming, though it did have certain preferences:

- “small languages” were thought desirable:
“Large languages present many problems (in implementation, training etc) and if a small language can describe the application succinctly it will be preferable.”
- Functional programming was liked, but with the comment:
“The absence of variables which are updated means that the exchange database has to be passed around as function arguments which is a bit awkward.”
- Logic programming was best in terms of elegance:
“Logic programming and the rule-based system gave the most unusual new approach to the problem with elegant solutions to some aspects of the problem.”
- Concurrency was viewed as essential for this type of problem, but:
“Adding concurrency to declarative languages, rule-based systems and the object based system is an open field for research.”

At this time, our experience with declarative languages was limited to PFL and LPL0, both developed in Sweden. PFL [17] came from the Programming Methodology Group at Chalmers Technical University in Gothenburg and was a version of ML extended with

²Plain Ordinary Telephone Service.

primitives borrowed from CCS. LPL0 [28] came from the Swedish Institute of Computer Science and was a logic language based on Haridi's natural deduction [15].

Looking back at the SPOTS paper, it is interesting to note what we weren't interested in—there is no mention in the paper of dealing with failure or providing fault-tolerance, there is no mention of changing the system as it is running or of how to make systems that scale dynamically. My own contribution to LOTS was to program POTS. This I did first in Smalltalk and then in Prolog. This was fairly sensible at the time, since I liberally interpreted Bjarne's directive to “solve all of Ericsson's software problems” as “program POTS in Smalltalk.”

3. Part II: 1985 – 1988. The birth of Erlang

3.1 Early experiments

My first attempts to make the phones ring was programmed in Smalltalk. I made a model with phone objects and an exchange object. If I sent a ring message to a phone it was supposed to ring. If the phone A went off-hook it was supposed to send an (offHook, A) message to the exchange. If the user of phone A dialled some digit D, then a (digit, A, D) message would be sent to the exchange.

Alongside the Smalltalk implementation, I developed a simple graphic notation that could be used to describe basic telephony. The notation describing telephony was then hand-translated into Smalltalk. By now the lab had acquired a SUN workstation with Smalltalk on it. But the Smalltalk was very slow—so slow that I used to take a coffee break while it was garbage collecting. To speed things up, in 1986 we ordered a Tektronix Smalltalk machine, but it had a long delivery time. While waiting for it to be delivered, I continued fiddling with my telephony notation. One day I happened to show Roger Skagervall my algebra—his response was “but that's a Prolog program.” I didn't know what he meant, but he sat me down in front of his Prolog system and rapidly turned my little system of equations into a running Prolog program. I was amazed. This was, although I didn't know it at the time, the first step towards Erlang.

My graphic notation could now be expressed in Prolog syntax and I wrote a report [1] about it. The algebra had predicates:

```
idle(N)    means the subscriber N is idle
on(N)     means subscribed N in on hook
...
```

And operators:

```
+t(A, dial_tone) means add a dial tone to A
```

Finally rules:

```
process(A, f) :- on(A), idle(A), +t(A,dial-tone),
                 +d(A, []), -idle(A), +of(A)
```

This had the following declarative reading:

```
process(A, f)      To process an off hook signal from
                   a subscriber A
:-                  then
on(A)               If subscriber A is on-hook
,                  and
idle(A)             If subscriber A is idle
,                  and
+t(A, dial_tone)   send a dial tone to A
,                  and
+d(A, [])          set the set of dialled digits to []
,                  and
-idle(A)           retract the idle state
,                  and
```

```
+of(A)           assert that we are off hook
```

Using this notation, POTS could be described using fifteen rules. There was just one major problem: the notation only described how one telephone call should proceed. How could we do this for thousands of simultaneous calls?

3.2 Erlang conceived

Time passed and my Smalltalk machine was delivered, but by the time it arrived I was no longer interested in Smalltalk. I had discovered Prolog and had found out how to write a meta-interpreter in Prolog. This meta-interpreter was easy to change so I soon figured out how to add parallel processes to Prolog. Then I could run several versions of my little telephony algebra in parallel.

The standard way of describing Prolog in itself is to use a simple meta-interpreter:

```
solve((A,B)) :- solve(A), solve(B).
solve(A)    :- builtin(A), call(A).
solve(A,B)  :- rule(A, B), solve(B).
```

The problem with this meta-interpreter is that the set of remaining goals that is not yet solved is not available for program manipulation. What we would like is a way to explicitly manage the set of remaining goals so that we could suspend or resume the computation at any time.

To see how this works, we can imagine a set of equations like this:

```
x -> a,b,c
a -> p,{q},r
r -> g,h
p -> {z}
```

This notation means that the symbol *x* is to be replaced by the sequence of symbols *a*, *b* and *c*. That *a* is to be replaced by *p*, *{q}* and *r*. Symbols enclosed in curly brackets are considered primitives that cannot be further reduced.

To compute the value of the symbol *x* we first create a stack of symbols. Our reduction machine works by successively replacing the top of the stack by its definition, or if it is a primitive by evaluating the primitive.

To reduce the initial goal *x* we proceed as follows:

<i>x</i>	replace <i>x</i> by its definition
<i>a,b,c</i>	replace <i>a</i> by its definition
<i>p,{q},r,b,c</i>	replace <i>p</i> by its definition
<i>{z},{q},r,b,c</i>	evaluate <i>z</i>
<i>{q},r,b,c</i>	evaluate <i>q</i>
<i>r,b,c</i>	replace <i>r</i> by its definition
<i>g,h,b,c</i>	...

The point of the reduction cycle is that at any time we can suspend the computation. So, for example, after three iterations, the state of the computation is represented by a stack containing:

```
{z},{q},r,b,c
```

If we want several parallel reduction engines, we arrange to save and store the states of each reduction engine after some fixed number of reductions. If we now express our equations as Prolog terms:

```
eqn(x,[a,b,c]).
eqn(a,[p,{q},r]).
eqn(r,[g,h]).
eqn(p,[{z}]).
```

Then we can describe our reduction engine with a predicate reduce as follows:

```
reduce([]).
reduce([{}|T]) :- call(H), !, reduce(T).
reduce([Lhs|More]) :- eqn(Lhs, Rhs),
append(Rhs, More, More1), !,
reduce(More1).
```

With a few more clauses, we can arrange to count the number of reduction steps we have made and to save the list of pending goals for later evaluation. This is exactly how the first Erlang interpreter worked. The interested reader can consult [4] for more details.

Time passed and my small interpreter grew more and more features, the choice of which was driven by a number of forces. First, I was using the emerging language to program a small telephone exchange, so problems naturally arose when I found that interacting with the exchange was clumsy or even impossible. Second, changes suggested themselves as we thought up more beautiful ways of doing things. Many of the language changes were motivated by purely aesthetic concerns of simplicity and generality.

I wanted to support not only simple concurrent processes, but also mechanisms for sending message between processes, and mechanism for handling errors, etc. My interpreter grew and some of the other lab members became interested in what I was doing. What started as an experiment in “*adding concurrency to Prolog*” became more of a language in its own right and this language acquired a name “Erlang,” which was probably coined by Bjarne Däcker. What did the name Erlang mean? Some said it meant “Ericsson Language,” while others claimed it was named after Agner Krarup Erlang (1878 – 1929), while we deliberately encouraged this ambiguity.

While I had been fiddling with my meta-interpreter, Robert Virding had been implementing variants of parallel logic programming languages. One day Robert came to me and said he’d been looking at my interpreter and was thinking about making a few small minor changes, did I mind? Now Robert is incapable of making small changes to anything, so pretty soon we had two different Erlang implementations, both in Prolog. We would take turns in rewriting each other’s code and improving the efficiency of the implementation.

As we developed the language, we also developed a philosophy around the language, ways of thinking about the world and ways of explaining to our colleagues what we were doing. Today we call this philosophy *Concurrency-Oriented Programming*. At the time our philosophy had no particular name, but was more just a set of rules explaining how we did things.

One of the earliest ways of explaining what Erlang was all about was to present it as a kind of hybrid language between concurrent languages and functional languages. We made a poster showing this which was reproduced in [12] and shown here in Figure 1.

3.3 Bollmora, ACS/Dunder

By 1987, Erlang was regarded as a new programming language that we had prototyped in Prolog. Although it was implemented in Prolog, Erlang’s error-handling and concurrency semantics differed significantly from Prolog. There were now two people (Robert Virding and I) working on the implementation and it was ready to be tried out on external users. By the end of the year, Mike Williams managed to find a group of users willing to try the language on a real problem, a group at *Ericsson Business Communications AB*, which was based in Bollmora. The group was headed by Kerstin Ödling and the other members of the team were Åke Rosberg,

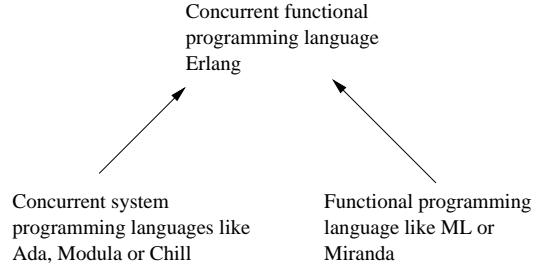


Figure 1. Relation of Erlang to existing languages.

Håkan Karlsson and Håkan Larsson. These were the first ever Erlang users.

The team wanted to prototype a new software architecture called ACS³ designed for programming telephony services on the Ericsson MD110 PABX⁴ and were looking for a suitable language for the project, which is how they got to hear about Erlang. A project called ACS/Dunder was started to build the prototype.

The fact that somebody was actually interested in what we were doing came as a great stimulus to the development and we entered a period of rapid development where we could actually try out our language ideas on real users.

3.4 Frenzied activity

Erlang began to change rapidly. We now had two people working on the implementation (Robert and myself) and a large user community (three people). We would add features to the language and then try them out on our users. If the users or implementors liked the changes, they stayed in. If the users disliked the changes or if the implementation was ugly, the changes were removed. Amazingly, the fact that the language was changing under their feet almost every day didn’t particularly bother our users. We met our Bollmora users once or twice a week for about six months. We taught them programming, they taught us telephony and both sides learned a lot.

Hardly anything remains from this period—most of the day-to-day changes to the language were not recorded and there is no lasting evidence of what those changes were. But fortunately, a few documents do remain: Figure 2 shows the entire Erlang 1.05 manual and Appendix A contains the comments at the start of the file erlang.pro (which was the main program for the Prolog interpreter). This is a change log documenting the changes made to the language in the nine-month period from 24 March 1988 to 14 December 1988. Like a child, Erlang took about nine months to develop. By the end of 1988, most of the ideas in Erlang had stabilized. While Robert and I implemented the system, the ideas behind the mechanism that we implemented often came from our users, or from the other members of the Lab. I always considered the morning coffee break to be the key forum where the brilliant ideas you had on the way to work were trashed and where all the real work was done. It was in these daily brainstorms that many a good idea was created. It’s also why nobody can quite remember who thought of what, since everybody involved in the discussions seems to remember that it was *they* who had the key idea.

It was during this period that most of the main ideas in Erlang emerged. In the following sections I will describe Erlang as it was in 1988. Appendix B has a set of examples illustrating the language as it is today.

³ Audio Communication System.

⁴ Private Automatic Branch Exchange.

erlang vsn 1.05

h	help
reset	reset all queues
reset_erlang	kill all erlang definitions
load(F)	load erlang file <F>.erlang
load	load the same file as before
load(?)	what is the current load file
what_erlang	list all loaded erlang files
go	reduce the main queue to zero
send(A,B,C)	perform a send to the main queue
send(A,B)	perform a send to the main queue
cq	see queue - print main queue
wait_queue(N)	print wait_queue(N)
cf	see frozen - print all frozen states
eqns	see all equations
eqn(N)	see equation(N)
start(Mod,Goal)	starts Goal in Mod
top	top loop run system
q	quit top loop
open_dots(Node)	opens Node
talk(N)	N=1 verbose, =0 silent
peep(M)	set peeping point on M
no_peep(M)	unset peeping point on M
vsn(X)	erlang vsn number is X

Figure 2. The Erlang 1.05 manual.

3.5 Buffered message reception

One of the earliest design decisions in Erlang was to use a form of buffering selective receive. The syntax for message reception in modern Erlang follows a basic pattern:

```
receive
    Pattern1 ->
        Actions1;
    Pattern2 ->
        Actions2;
    ...
end
```

This means wait for a message. If the message matches **Pattern1** then evaluate the code **Actions1**. If the message matches **Pattern2** then evaluate the code **Actions2**, etc.

But what happens if some other message that matches neither **Pattern1** or **Pattern2** arrives at the processes? Answer—ignore the message and queue it for later. The message is put in a “save queue” and processed later and the **receive** statement carries on waiting for the messages it is interested in.

The motivation for automatically buffering out-of-order messages came from two observations. Firstly we observed that in the CCITT Specification and Description Language⁵ (SDL), one of the most commonly used specification idioms involved queuing and later replaying out-of-order messages. Secondly we observed that handling out-of-order messages in a finite state machine led to an explosion in the state space of the finite state machine. This happens more often than you think, in particular when processing remote procedure calls. Most of the telecommunications programs we write deal with message-oriented protocols. In implementing the protocols we have to handle the messages contained in the protocol itself together with a large number of messages that are not

⁵ SDL is widely used in the telecoms industry to specify communications protocols.

part of the protocol but come from remote procedure calls made internally in the system. Our queuing strategy allows us to make an internal remote procedure call within the system and block until that call has returned. Any messages arriving during the remote procedure call are merely queued and served after the remote procedure call has completed. The alternative would be to allow the possibility of handling protocol messages in the middle of a remote procedure call, something which greatly increases the complexity of the code.

This was a great improvement over PLEX, where every message must be handled when it arrives. If a messages arrives “too early” the program has to save it for later. Later, when it expects the message, it has to check to see if the message has already arrived.

This mechanism is also extremely useful for programming sets of parallel processes when you don’t know in which order the message between the processes will arrive. Suppose you want to send three messages M1, M2 and M3 to three different processes and receive replies R1, R2 and R3 from these three processes. The problem is that the reply messages can arrive in any order. Using our receive statement, we could write:

```
A ! M1,
B ! M2,
C ! M3,
receive
    A ? R1 ->
    receive
        B ? R2 ->
        receive
            C ? R3 ->
            ... now R1 R2 and R3
            have been received ...
```

Here **A ! M** means send the message **M** to the process **A** and **A ? X** means receive the message **X** from the process **A**.

It doesn’t matter now in which order the messages are received. The code is written as if **A** replies first—but if the message from **B** replies first, the message will be queued and the system will carry on waiting for a message from **A**. Without using a buffer, there are six different orderings of the message to be accounted for.

3.6 Error handling

Error handling in Erlang is very different from error handling in conventional programming languages. The key observation here is to note that the error-handling mechanisms were designed for building fault-tolerant systems, and not merely for protecting from program exceptions. You cannot build a fault-tolerant system if you only have one computer. The minimal configuration for a fault-tolerant system has two computers. These must be configured so that both observe each other. If one of the computers crashes, then the other computer must take over whatever the first computer was doing.

This means that the model for error handling is based on the idea of two computers that observe each other. Error detection and recovery is performed on the remote computer and not on the local computer. This is because in the worst possible case, the computer where the error has occurred has crashed and thus no further computations can be performed on the crashed computer.

In designing Erlang, we wanted to abstract all hardware as reactive objects. Objects should have “process semantics;” in other words, as far as the software was concerned, the only way to interact with hardware was through message passing. When you send a message to a process, there should be no way of knowing if the process was really some hardware device or just another software process. The reason for this was that in order to simplify our programming model, we wanted to model everything as processes and

we wanted to communicate with all processes in a uniform manner. From this point of view we wanted software errors to be handled in exactly the same manner as hardware errors. So, for example, if a process died because of a divide by zero it would propagate an `{'EXIT', Pid, divideByZero}` signal to all the processes in its link set. If it died because of a hardware error it might propagate an `{'EXIT', Pid, machineFailure}` signal to its neighbors. From a programmer's point of view, there would no difference in how these signals were handled.

3.7 Links

Links in Erlang are provided to control error propagation paths for errors between processes. An Erlang process will die if it evaluates illegal code, so, for example, if a process tries to divide by zero it will die. The basic model of error handling is to assume that some other process in the system will observe the death of the process and take appropriate corrective actions. But which process in the system should do this? If there are several thousand processes in the system then how do we know which process to inform when an error occurs? The answer is the *linked* process. If some process A evaluates the primitive `link(B)` then it becomes linked to A. If A dies then B is informed. If B dies then A is informed.

Using links, we can create sets of processes that are linked together. If these are normal⁶ processes, they will die immediately if they are linked to a process that dies with an error. The idea here is to create sets of processes such that if any process in the set dies, then they will all die. This mechanism provides the invariant that either all the processes in the set are alive or none of them are. This is very useful for programming error-recovery strategies in complex situations. As far as I know, no other programming language has anything remotely like this.

The idea of links and of the mechanism by which all processes in a set die was due to Mike Williams. Mike's idea was inspired by the design of the release mechanism used in old analogue telephones and exchanges. In the analogue telephones and in the early electromechanical exchanges, three wires called A, B and C were connected to the phones. The C wire went back to the exchange and through all the electromechanical relays involved in setting up a call. If anything went wrong, or if either partner terminated the call, then the C wire was grounded. Grounding the C wire caused a knock-on effect in the exchange that freed all resources connected to the C line.

3.8 Buffers

Buffers and the `receive` statement fit together in a rather non-obvious way. Messages sent between Erlang processes are always delivered as soon as possible. Each process has a "mailbox" where all incoming messages are stored. When a message arrives it is put in the mailbox and the process is scheduled for execution. When the process is next scheduled it tries to pattern match the message. If the match succeeds, message reception occurs and the message is removed from the mailbox and the data from the message works its way into the program. Otherwise the message is put into a "save" queue. When any message matches, the entire save queue is merged back into the mailbox. All buffering that takes place is then performed implicitly in either the mailbox or the save queue. In about 1988, this mechanism was the subject of intense debate. I remember something like a four-day meeting being devoted to the single topic of how interprocess communication should actually work.

The outcome of this meeting was that we all thought that processes should communicate through pipes and that the pipes should

⁶ Process are either normal or system processes. System process can trap and process non-normal exit signals. Normal process just die.

be first-class objects. They should have infinite⁷ buffering capacity, they should be named and it should be possible to connect and disconnect them to and from processes. It should be possible to bend them and split them and join them, and I even created an algebra of pipes.

Then I tried to implement the pipe algebra but this turned out to be very difficult. In particular, pipe joining and coalescing⁸ operations were terribly difficult to program. The implementation needed two types of message in the pipes: regular messages and small tracer messages that had to be sent up and down the pipes to check that they were empty. Sometimes the pipes had to be locked for short periods of time and what would happen if the processes at the end of the pipes failed was extremely difficult to work out. The problems all stem from the fact that the pipes introduce dependencies between the end points so that the processes at either end are no longer independent—which makes life difficult.

After two weeks of programming, I declared that the pipe mechanism now worked. The next day I threw it all away—the complexity of the implementation convinced me that the mechanism was wrong. I then implemented a point-to-point communication mechanism with mailboxes; this took a couple of hours to implement and suffered from none of the kind of problems that plagued the pipes implementation. This seems to be a rather common pattern: first I spend a week implementing something, then, when is is complete I throw everything away and reimplement something slightly different in a very short time.

At this point, pipes were rejected and mailboxes accepted.

3.9 Compilation to Strand

While the Prolog implementations of Erlang were being used to develop the language and experiment with new features, another avenue of research opened, work aimed at creating an efficient implementation of Erlang. Robert had not only implemented Erlang in Prolog but had also been experimenting with a Parlog compiler of his own invention. Since Erlang was a concurrent language based on Prolog it seemed natural to study how a number of concurrent logical programming languages had been implemented. We started looking at languages like Parlog, KL1 and Strand for possible inspiration. This actually turned out to be a mistake. The problem here has to do with the nature of the concurrency. In the concurrent logic programming languages, concurrency is implicit and extremely fine-grained. By comparison Erlang has explicit concurrency (via processes) and the processes are coarse-grained. The study of Strand and Parlog led to an informal collaboration with Keith Clarke and Ian Foster at Imperial College London.

Eventually, in 1988, we decided to experiment with cross compilation of Erlang to Strand. Here we made a major error—we confidently told everybody the results of the experiment before we had done it. Cross compilation to Strand would speed up Erlang by some embarrassingly large factor. The results were lacklustre; the system was about five times faster but very unstable and large programs with large numbers of processes just would not run. The problem turned out to be that Strand was just too parallel and created far too many parallel processes. Even though we might only have had a few hundred Erlang processes, several tens of millions of parallel operations were scheduled within the Strand system.

Strand also had a completely different model of error handling and message passing. The Erlang-to-Strand compiler turned an Erlang function of arity N to a Strand process definition of arity N+8. The eight additional arguments were needed to implement Erlang's

⁷ In theory.

⁸ For example, if pipe X has endpoints A and B, and pipe Y has endpoints C and D, then coalescing X and Y was performed with an operation `muff_pipes(B, C)`.

error-handling and message-passing semantics. More details of this can be found in chapter 13 of [14]. The problem with the implementation boiled down to the semantic mismatch between the concurrency models used in Strand and Erlang, which are completely different.

4. Part III: 1989 – 1997. Erlang grows

The eight years from 1989 to 1997 were the main period during which Erlang underwent a period of organic growth. At the start of the period, Erlang was a two-man project, by the end hundreds of people were involved.

The development that occurred during this period was a mixture of frantic activity and long periods where nothing appeared to happen. Sometimes ideas and changes to the language came quickly and there were bursts of frenetic activity. At other times, things seem to stagnate and there was no visible progress for months or even years. The focus of interest shifted in an unpredictable manner from technology to projects, to training, to starting companies. The dates of particular events relating to the organization, like, for example, the start of a particular project or the formation of Erlang Systems AB, are wellknown. But when a particular idea or feature was introduced into the language is not so wellknown. This has to do with the nature of software development. Often a new software feature starts as a vague idea in the back of implementors' minds, and the exact date when they had the idea is undocumented. They might work on the idea for a while, then run into a problem and stop working on it. The idea can live in the back of their brain for several years and then suddenly pop out with all the details worked out.

The remainder of this section documents in roughly chronological order the events and projects that shaped the Erlang development.

4.1 The ACS/Dunder results

In December 1989, the ACS/Dunder project produced its final report. For commercial reasons, this report was never made public. The report was authored by the members of the prototyping team who had made the ACS/Dunder prototype, in which about 25 telephony features were implemented. These features represented about one tenth of the total functionality of the MD 110. They were chosen to be representative of the kind of features found in the MD110, so they included both hard and extremely simple functions. The report compared the effort (in man hours) of developing these features in Erlang with the predicted effort of developing the same features in PLEX. The ACS/Dunder report found that the time to implement the feature in Erlang divided by the time to implement the feature in PLEX (measured in man hours) was a factor of 3 to 25, depending upon the feature concerned. The average increase in productivity was a factor of 8.

This factor and the conclusion of the report were highly controversial and many theories were advanced to explain away the results. It seemed at the time that people disliked the idea that the effect could be due to having a better programming language, preferring to believe that it was due to some “smart programmer effect.” Eventually we downgraded the factor to a mere 3 because it sounded more credible than 8. The factor 3 was totally arbitrary, chosen to be sufficiently high to be impressive and sufficiently low to be believable. In any case, it was significantly greater than one, no matter how you measured and no matter how you explained the facts away.

The report had another conclusion, namely that Erlang was far too slow for product development. In order to use Erlang to make a real product, it would need to be at least 40 times faster. The fact that it was too slow came from a comparison of the execution times of the Erlang and PLEX programs. At this stage, CPU per-

formance represented the only significant problem. Memory performance was not a problem. The run-time memory requirements were modest and the total size of the compiled code did not pose any problems.

After a lot of arguing, this report eventually led the way to the next stage of development, though the start of the project was to be delayed for a couple of years. Ericsson decided to build a product called the *Mobility Server* based on the ACS/Dunder architecture, and we started work on a more efficient implementation of Erlang.

4.2 The word starts spreading

1989 also provided us with one of our first opportunities to present Erlang to the world outside Ericsson. This was when we presented a paper at the SETSS conference in Bournemouth. This conference was interesting not so much for the paper but for the discussions we had in the meetings and for the contacts we made with people from Bellcore. It was during this conference that we realised that the work we were doing on Erlang was very different from a lot of mainstream work in telecommunications programming. Our major concern at the time was with detecting and recovering from errors. I remember Mike, Robert and I having great fun asking the same question over and over again: “*what happens if it fails?*”—the answer we got was almost always a variant on “*our model assumes no failures.*” We seemed to be the only people in the world designing a system that could recover from software failures.

It was about this time that we realized very clearly that shared data structures in a distributed system have terrible properties in the presence of failures. If a data structure is shared by two physical nodes and if one node fails, then failure recovery is often impossible. The reason why Erlang shares no data structures and uses pure copying message passing is to sidestep all the nasty problems of figuring out what to replicate and how to cope with failures in a distributed system. At the Bournemouth conference everybody told us we were wrong and that data must be shared for efficiency—but we left the conference feeling happy that the rest of the world was wrong and that we were right. After all, better a slow system that can recover from errors than a fast system that cannot handle failures. Where people were concerned with failure, it was to protect themselves from *hardware failures*, which they could do by replicating the hardware. In our world, we were worried by *software failures* where replication does not help.

In Bournemouth, we met a group of researchers headed by Gary Herman from Bellcore who were interested in what we were doing. Later in the year, in December 1989, this resulted in Bjarne, Mike, Robert and me visiting Bellcore, where we gave our first ever external Erlang lecture. Erlang was well received by the researchers at Bellcore, and we were soon involved in discussing how they could get a copy of the Erlang system. When we got back from Bellcore, we started planning how to release Erlang. This took a while since company lawyers were involved, but by the middle of 1990 we delivered a copy of Erlang to Bellcore. So now we had our first external user, John Unger from Bellcore.

4.3 Efficiency needed – the JAM

Following the ACS/Dunder report in December 1989, we started work on an efficient version of Erlang. Our goal was to make a version of Erlang that was at least 40 times faster than the prototype.

At this point we were stuck. We knew what we wanted to do but not how to do it. Our experiments with Strand and Parlog had led to a deadend. Since we were familiar with Prolog, the next step appeared to follow the design of an efficient Prolog machine. Something like the WAM [29] seemed the natural way to proceed. There were two problems with this. First, the WAM didn’t support concurrency and the kind of error handling that we were interested

in, and second, we couldn't understand how the WAM worked. We read all the papers but the explanations seemed to be written only for people who already understood how the thing worked.

The breakthrough came in early 1990. Robert Virding had collected a large number of the descriptions of abstract machines for implementing parallel logic machines. One weekend I borrowed his file and took all the papers home. I started reading them, which was pretty quick since I didn't really understand them, then suddenly after I'd read through them all I began to see the similarities. A clue here, and hint there, yes they were all the same. Different on the surface, but very similar underneath. I understood—then I read them again, this time slowly. What nobody had bothered to mention, presumably because it was self-evident, was that each of the instructions in one of these abstract machines simultaneously manipulated several registers and pointers. The papers often didn't mention the stack and heap pointers and all the things that got changed when an instruction was evaluated because this was obvious.

Then I came across a book by David Maier and David Scott Warren [19] that confirmed this; now I could understand how the WAM worked. Having understood this, it was time to design my own machine, the JAM.⁹ Several details were missing from the Warren paper and from other papers describing various abstract machines. How was the code represented? How was the compiler written? Once you understood them, the papers seem to describe the easy parts; the details of the implementation appeared more to be “trade secrets” and were not described.

Several additional sources influenced the final design. These included the following:

- A portable Prolog Compiler [9], that described how virtual machine instructions were evaluated.
- A Lisp machine with very compact programs [13], that described a Lisp machine with an extremely compact representation.
- BrouHaHa – A portable Smalltalk interpreter [22], that described some smart tricks for speeding up the dispatching instructions in a threaded interpreter.

When I designed the JAM, I was worried about the expected size of the resulting object for the programs. Telephony control programs were huge, numbering tens of millions of lines of source code. The object code must therefore be highly compact, otherwise the entire program would never fit into memory. When I designed the JAM, I sat down with my notepad, invented different instruction sets, then wrote down how some simple functions could be compiled into these instruction sets. I worked out how the instructions would be represented in a byte-coded machine and then how many instructions would be evaluated in a top-level evaluation of the function. Then I changed the instruction set and tried again. My benchmark was always for the “append” function and I remember the winning instruction set compiled append into 19 bytes of memory. Once I had decided on an instruction set, compilation to the JAM was pretty easy and an early version of the JAM is described in [5]. Figure 3 shows how the factorial function was compiled to JAM code.

Being able to design our own virtual machines resulted in highly compact code for the things we cared most about. We wanted message passing to be efficient, which was easy. The JAM was a stack-based machine so to compile `A ! B`, the compiler emitted code to compile `A`, then code to compile `B`, then a single byte send instruction. To compile `spawn(Function)`, the compiler emitted code to build a function closure on the stack, followed by a single byte spawn instruction.

⁹ Modesty prevents me from revealing what this stands for.

```

fac(0) -> 1;
fac(N) -> N * fac(N-1)

{info, fac, 1}
{try_me_else, label1}
{arg, 0}
{getInt, 0}
{pushInt, 1}
ret
label1: try_me_else_fail
{arg, 0}
dup
{pushInt, 1}
minus
{callLocal, fac, 1}
times
ret

```

Figure 3. Compilation of sequential Erlang to JAM code.

File	Lines	Purpose
sys.sys.erl	18	dummy
sys.parse.erl	783	erlang parser
sys.ari_parser.erl	147	parse arithmetic expressions
sys.build.erl	272	build function call arguments
sys.match.erl	253	match function head arguments
sys.compile.erl	708	compiler main program
sys.lists.erl	85	list handling
sys.dictionary.erl	82	dictionary handler
sys.utils.erl	71	utilities
sys.asm.erl	419	assembler
sys.tokenise.erl	413	tokeniser
sys.parser_tools.erl	96	parser utilities
sys.load.erl	326	loader
sys.opcodes.erl	128	opcode definitions
sys.pp.erl	418	pretty printer
sys.scan.erl	252	scanner
sys.boot.erl	59	bootstrap
sys.kernel.erl	9	kernel calls
18 files	4544	

Table 2. Statistics from an early Erlang compiler.

The compiler was, of course, written in Erlang and run through the Prolog Erlang emulator. To test the abstract machine I wrote an emulator, this time in Prolog so I could now test the compiler by getting it to compile itself. It was not fast—it ran at about 4 Erlang reductions¹⁰ per second, but it was fast enough to test itself. Compiling the compiler took a long time and could only be done twice a day, either at lunch or overnight.

The compiler itself was a rather small and simple program. It was small because most of the primitives in Erlang could be compiled into a single opcode in the virtual machine. So all the compiler had to do was to generate code for efficient pattern matching and for building and reconstructing terms. Most of the complexity is in the run-time system, which implements the opcodes of the virtual machine. The earliest compiler I have that has survived is the erl89 compiler, which had 18 modules containing 2544 lines of code. The modules in the compiler were as in Table 2.

It was now time to implement the JAM virtual machine emulator, this time not in Prolog but in C. This is where Mike Williams

¹⁰One reduction corresponds to a single function call.

came in. I started writing the emulator myself in C but soon Mike interfered and started making rude comments about my code. I hadn't written much C before and my idea of writing C was to close my eyes and pretend it was FORTRAN. Mike soon took over the emulator, threw away all my code and started again. Now the Erlang implementor group had expanded to three, Mike, Robert and myself. Mike wrote the inner loop of the emulator very carefully, since he cared about the efficiency of the critical opcodes used for concurrent operations. He would compile the emulator, then stare at the generated assembler code, then change the code compile again, and stare at the code until he was happy. I remember him working for several days to get message sending just right. When the generated code got down to six instructions he gave up.

Mike's emulator soon worked. We measured how fast it was. After some initial tweaking, it ran 70 times faster than the original Prolog emulator. We were delighted—we had passed our goal of 40 by a clear margin. Now we could make some real products.

Meanwhile, the news from the Bollmora group was not good: "We miscalculated, our factor of 40 was wrong, it needs to be 280 times faster."

One of the reviewers of this paper asked whether memory efficiency was ever a problem. At this stage the answer was no. CPU efficiency was always a problem, but never memory efficiency.

4.4 Language changes

Now that we have come to 1990 and have a reasonably fast Erlang, our attention turned to other areas. Efficiency was still a problem, but spurred by our first success we weren't particularly worried about this. One of the things that happened when writing Erlang in Erlang was that we had to write our own parser. Before we had just used infix operators in Prolog. At this point the language acquired its own syntax and this in its turn caused the language to change. In particular, `receive` was changed.

Having its own syntax marked a significant change in the language. The new version of Erlang behaved pretty much like the old Prolog interpreter, but somehow it *felt* different. Also our understanding of the system deepened as we grappled with tricky implementation issues that Prolog had shielded us from. In the Prolog system, for example, we did not have to bother about garbage collection, but in our new Erlang engine we had to implement a garbage collector from scratch.

Since our applications ran in the so-called soft real-time domain, the performance of the garbage collector was crucial, so we had to design and implement garbage-collection strategies that would not pause the system for too long. We wanted frequent small garbage collections rather than infrequent garbage collections that take a long time.

The final strategy we adopted after experimenting with many different strategies was to use per-process stop-and-copy GC. The idea was that if we have many thousands of small processes then the time taken to garbage collect any individual process will be small. This strategy also encouraged copying all the data involved in message passing between processes, so as to leave no dangling pointers between processes that would complicate garbage collection. An additional benefit of this, which we didn't realise at the time, was that copying data between processes increases process isolation, increases concurrency and simplifies the construction of distributed systems. It wasn't until we ran Erlang on multicore CPUs that the full benefit of non-shared memory became apparent. On a multicore CPU, message passing is extremely quick and the lack of locking between CPUs allows each CPU to run without waiting for the other CPUs.

Our approach to GC seemed a little bit reckless: would this method work in practice? We were concerned about a number of problems. Would large numbers of processes decide to garbage col-

```
# wait_first_digit(A) ->
    receive 10 {
        A ? digit(D) =>
            stop_tone(A),
            received_digit(A, [], D);
        A ? on_hook =>
            stop_tone(A),
            idle(A);
        timeout =>
            stop_tone(A),
            wait_clear(A);
        Other =>
            wait_first_digit(A)
    }.
```

Erlang in 1988

```
wait_first_digit(A) ->
    receive
        {A, {digit, D}} ->
            stop_tone(A),
            received_digit(A, [], D);
        {A, on_hook} ->
            stop_tone(A),
            idle(A);
        Other ->
            wait_first_digit(A)
    after 10 ->
        stop_tone(A),
        wait_clear(A)
    end.
```

Erlang today

Figure 4. Erlang in 1988 and today.

lect all at the same time? Would programmers be able to structure their applications using many small processes, or would they use one large process? If they did use one large process, what would happen when it performed a garbage collection? Would the system stop? In practice our fears were unfounded. Process garbage collections seem to occur at random and programmers very rarely use a single large process to do everything. Current systems run with tens to hundreds of thousands of processes and it seems that when you have such large numbers of processes, the effects of GC in an individual process are insignificant.

4.5 How receive changed and why

The early syntax of Erlang came straight from Prolog. Erlang was implemented directly in Prolog using a careful choice of infix operators. Figure 4 adapted from [2] shows a section of a telephony program from 1988 and the corresponding program as it would be written today. Notice there are two main changes:

First, in the 1988 example, patterns were represented by Prolog terms, thus `digit(D)` represents a pattern. In modern Erlang, the same syntax represents a function call and the pattern is written `{digit,D}`.

The second change has to do with how message reception patterns were written. The syntax:

```
Proc ! Message
```

means send a message, while:

```
receive {
    Proc1 ? Mess1 =>
        Actions1;
```

```

rpc(Pid, Query) ->
    Pid ! {self(), Query},
    receive
        {Pid, Reply} ->
            Reply
    end.

```

Client code

```

server(Data) ->
    receive
        {From, Query} ->
            {Reply, Data1} = F(Query, Data),
            From ! {self(), Reply},
            server(Data1)
    end.

```

Server Code

Figure 5. Client and server code for a remote procedure call.

```

Proc2 ? Mess2 =>
    Actions2;
    ...
}

```

means try to receive a message `Mess1` from `Proc1`, in which case perform `Actions1`; otherwise try to receive `Mess2` from `Proc2`, etc. The syntax `Proc?Message` seemed at first the obvious choice to denote message reception and was mainly chosen for reasons of symmetry. After all if `A!B` means send a message then surely `A?B` should mean receive a message.

The problem with this is that there is no way to hide the identity of the sender of a message. If a process A sends a message to B, then receiving the message with a pattern of the form `P?M` where P and M are unbound variables always results in the identity of A being bound to P. Later we decided that this was a bad idea and that the sender should be able to choose whether or not it reveals its identity to the process that it sends a message to. Thus if A wants to send a message M to a process B and reveal its identity, we could write the code which sends a message as `B!{self(), M}` or, if it did not wish to reveal its identity, we could write simply `B!M`. The choice is not automatic but is decided by the programmer.

This leads to the common programming idiom for writing a remote procedure call whereby the sender must always include its own `Pid`¹¹ (the `Pid` to reply to) in the message sent to a server. The way we do this today is shown in Figure 9.

Note that we can also use the fact that processes do not reveal their identity to write secure code and to fake the identity of a process. If a message is sent to a process and that message contains no information about the sender, then there is no way the receiver of the message can know from whom the message was sent. This can be used as the basis for writing secure code in Erlang.

Finally, faked message `Pids` can be used for delegation of responsibilities. For example, much of the code in the IO subsystem is written with `{Request, ReplyTo, ReplyAs}` messages, where `Request` is a term requesting some kind of IO service. `ReplyTo` and `ReplyAs` are `Pids`. When the final process to perform the operation has finished its job, it sends a message by evaluating `ReplyTo ! {ReplyAs, Result}`. If this is then used in code in the RPC programming idiom in Figure 5, the code essentially fakes the `Pid` of the responding process.

Now the point of all this argument, which might seem rather obscure, is that a seemingly insignificant change to the surface

syntax, i.e. breaking the symmetry between `A!B` and `A?B`, has profound consequences on security and how we program. And it also explains the hour-long discussions over the exact placement of commas and more importantly what they mean.¹²

4.6 Years pass ...

The next change was the addition of distribution to the language. Distribution was always planned but never implemented. It seemed to us that adding distribution to the language would be easy since all we had to do was add message passing to remote processes and then everything should work as before.

At this time, we were only interested in connecting conventional sequential computers with no shared memory. Our idea was to connect stock hardware through TCP/IP sockets and run a cluster of machines behind a corporate firewall. We were not interested in security since we imagined all our computers running on a private network with no external access. This architecture led to a form of all-or-nothing security that makes distributed Erlang suitable for programming cluster applications running on a private network, but unsuitable for running distributed applications where various degrees of trust are involved.

1990

In 1990 Claes (Klacke) Wikström joined the Lab—Klacke had been working in another group at Ellemtel and once he became curious about what we were doing we couldn't keep him away. Klacke joined and the Erlang group expanded to four.

ISS'90

One of the high points of 1990 was ISS'90 (International Switching Symposium), held in Stockholm. ISS'90 was the first occasion where we actively tried to market Erlang. We produced a load of brochures and hired a stall at the trade fair and ran round-the-clock demonstrations of the Erlang system. Marketing material from this period is shown in Figures 6 and 7.

At this time, our goal was to try and spread Erlang to a number of companies in the telecoms sector. This was viewed as strategically important—management had the view that if we worked together with our competitors on research problems of mutual interest, this would lead to successful commercial alliances. Ericsson never really had the goal of making large amounts of money by selling Erlang and did not have an organisation to support this goal, but it was interested in maintaining a high technical profile and interacting with like-minded engineers in other companies.

1991

In 1991, Klacke started work on adding distribution to Erlang, something that had been waiting to be done for a long time. By now, Erlang had spread to 30 sites. The mechanisms for this spread are unclear, but mostly it seems to have been by word-of-mouth. Often we would get letters requesting information about the system and had no idea where they had heard about it. One likely mechanism was through the usenet mailing lists where we often posted to `comp.lang.functional`. Once we had established a precedent of releasing the system to Bellcore, getting the system to subsequent users was much easier. We just repeated what we'd done for Bellcore. Eventually after we had released the system to a dozen or so users, our managers and lawyers got fed up with our pestering and let us release the system to whomever we felt like, provided they signed a non-disclosure agreement.

¹²All language designers are doubtless familiar with the phenomenon that users will happily discuss syntax for hours but mysteriously disappear when the language designer wants to talk about what the new syntax actually means.

¹¹Process Identifier.



Figure 6. Early internal marketing – the relationship between Erlang and PLEX.

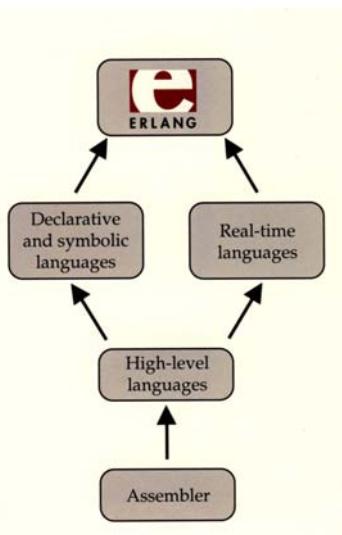


Figure 7. Erlang marketing – the relation of Erlang to other languages.

Also, Ericsson Business Communications ported Erlang to the FORCE computer and real-time OS VxWorks; these were our first steps towards an embedded system. This port was driven by product requirements since the mobility server at the time ran on VxWorks. We also ported Erlang to virtually all operating systems we had access to. The purpose of these ports was to make the language accessible to a large set of users and also to improve the quality of the Erlang system itself. Every time we ported the system to a new OS we found new bugs that emerged in mysterious ways, so porting to a large number of different OSs significantly improved the quality of the run-time system itself.

1992

In 1992, we got permission to publish a book and it was decided to commercialize Erlang. A contract was signed with Prentice Hall and the first Erlang book appeared in the bookshops in May 1993.

Even this decision required no small measure of management persuasion—this was definitely not how Ericsson had done things in the past; earlier languages like PLEX had been clothed in se-

crecy. Management's first reaction at the time was “if we've done something good, we should keep quiet about it”, quite the opposite to today's reaction to the open-source movement.

The decision to publish a book about Erlang marked a change in attitude inside Ericsson. The last language developed by Ericsson for programming switches was PLEX. PLEX was proprietary: very few people outside Ericsson knew anything about PLEX and there were no PLEX courses in the universities and no external market for PLEX programs or programmers. This situation had advantages and disadvantages. The major advantage was that PLEX gave Ericsson a commercial advantage over its competitors, who were presumed to have inferior technologies. The disadvantages had to do with isolation. Because nobody else used PLEX, Ericsson had to maintain everything to do with PLEX: write the compilers, hold courses, everything.

AT&T, however, had taken a different approach with C and C++. Here, the burden of supporting these languages was shared by an entire community and isolation was avoided. The decision to publish an Erlang book and to be fairly open about what we did was therefore to avoid isolation and follow the AT&T/C path rather than the Ericsson/PLEX path. Also in 1992 we ported Erlang to MS-DOS windows, the Mac, QNX and VxWorks.

The Mobility Server project, which was based upon the successful ACS/Dunder study, was started about two years after the ACS/Dunder project finished. Exactly why the mobility server project lost momentum is unclear. But this is often the way it happens: periods of rapid growth are followed by unexplained periods when nothing much seems to happen. I suspect that these are the consolidation periods. During rapid growth, corners get cut and things are not done properly. In the periods between the growth, the system gets polished. The bad bits of code are removed and reworked. On the surface not much is happening, but under the surface the system is being re-engineered.

1993

In May, the Erlang book was published.

4.7 Turbo Erlang

In 1993, the Turbo Erlang system started working. Turbo Erlang was the creation of Bogumil (Bogdan) Hausman who joined the Lab from SICS.¹³ For bizarre legal reasons the name Turbo Erlang was changed to BEAM.¹⁴ The BEAM compiler compiled Erlang programs to BEAM instructions.

The BEAM instructions could either be macro expanded into C and subsequently compiled or transformed into instructions for a 32-bit threaded code interpreter. BEAM programs compiled to C ran about ten times faster than JAM interpreted programs, and the BEAM interpreted code ran more than three times faster than the JAM programs.

Unfortunately, BEAM-to-C compiled programs increased code volumes, so it was not possible to completely compile large applications into C code. For a while we resolved this by recommending a hybrid approach. A small number of performance-critical modules would be compiled to C code while others would be interpreted.

The BEAM instruction set used fixed-length 32-bit instructions and a threaded interpreter, as compared to the JAM, which had variable length instructions and was a byte-coded interpreter. The threaded BEAM code interpreter was much more efficient than the JAM interpreter and did not suffer from the code expansion that compilation to C involved. Eventually the BEAM instruction set and threaded interpreter was adopted and the JAM phased out.

¹³ Swedish Institute of Computer Science.

¹⁴ Bogdan's Erlang Abstract Machine.

I should say a little more about code size here. One of the main problems in many products was the sheer volume of object code. Telephone switches have millions of lines of code. The current software for the AXD301, for example, has a couple of millions lines of Erlang and large amounts of sourced C code. In the mid-'90s when these products were developed, on-board memory sizes were around 256 Mbytes. Today we have Gbyte memories so the problem of object code volume has virtually disappeared, but it was a significant concern in the mid-'90s. Concerns about object-code memory size lay behind many of the decisions made in the JAM and BEAM instruction sets and compilers.

4.8 Distributed Erlang

The other major technical event in 1993 involved Distributed Erlang. Distribution has always been planned but we never had time to implement it. At the time all our products ran on single processors and so there was no pressing need to implement distribution. Distribution was added as part of our series of ongoing experiments, and wasn't until 1995 and the AXD301 that distribution was used in a product.

In adding distribution, Klacke hit upon several novel implementation tricks. One particularly worthy of mention was the atom communication cache described in [30], which worked as follows:

Imagine we have a distributed system where nodes on different hosts wish to send Erlang atoms to each other. Ideally we could imagine some global hash table, and instead of sending the textual representation of the atom, we would just send the atom's hashed value. Unfortunately, keeping such a hash table consistent is a hard problem. Instead, we maintain two synchronised hash tables, each containing 256 entries. To send an atom we hashed the atom to a single byte and then looked this up in the local hash table. If the value was in the hash table, then all we needed to do was to send to the remote machine a single byte hash index, so sending an atom between machines involved sending a single byte. If the value was not in the hash table, we invalidated the value in the cache and sent the textual representation of the atom.

Using this strategy, Klacke found that 45% of all objects sent between nodes in a distributed Erlang system were atoms and that the hit rate in the atom cache was around 95%. This simple trick makes Erlang remote procedure calls run somewhat faster for complex data structures than, for example, the SunOS RPC mechanism.

In building distributed Erlang, we now had to consider problems like dropped messages and remote failures. Erlang does not guarantee that messages are delivered but it does provide weaker guarantees on message ordering and on failure notification.

The Erlang view of the world is that message passing is unreliable, so sending a message provides no guarantee that it will be received. Even if the message were to be received, there is no guarantee that the message will be acted upon as you intended. We therefore take the view that if you want confirmation that a message has been received, then the receiver must send a reply message and you will have to wait for this message. If you don't get this reply message, you won't know what happened. In addition to this there is a link mechanism, which allows one process to link to another. The purpose of the link is to provide an error monitoring mechanism. If a process that you are linked to dies, you will be sent an error signal that can be converted to an error message.

If you are linked to a process and send a stream of messages to that process, it is valid to assume that no messages will be dropped, that the messages are not corrupted and that the messages will arrive at that process in the order they were sent or that an error has occurred and you will be sent an error signal. All of this presumes that TCP/IP is itself reliable, so if you believe that

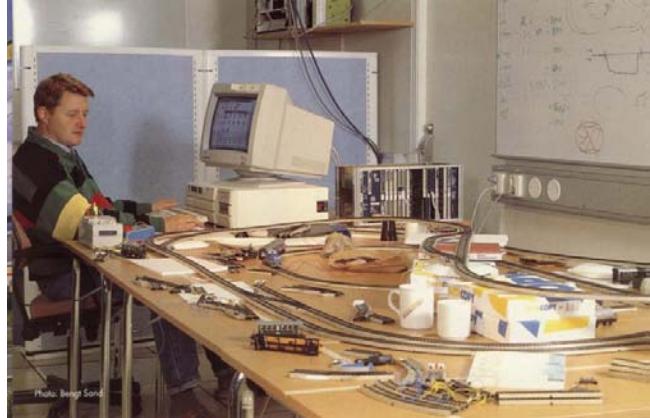


Figure 8. Robert Virding hard at work in the lab (1993).

TCP/IP is reliable then message passing between linked processes is reliable.

4.9 Spreading Erlang

In April 1993, a new company called *Erlang Systems AB* was formed, which was owned by *Ericsson Programatic*. The goal was to market and sell Erlang to external companies. In addition, Erlang Systems was to take over responsibility for training and consulting and the production of high-quality documentation.

Erlang Systems also provided the main source of employment for the "Uppsala boys". These were former computer science students from the University of Uppsala who had completed their Master's thesis studies with an Erlang project. Many of these students started their careers in Erlang Systems and were subsequently hired out to Ericsson projects as internal consultants. This proved a valuable way of "kick-starting" a project with young and enthusiastic graduate students who were skilled in Erlang.

Another memorable event of 1993 was the Erlang display at the trade fair held in October in Stockholm. The main demonstrator at the display was a program which simultaneously controlled a small telephone exchange¹⁵ and a model train. Figure 8 shows Robert Virding hard at work in the Lab programming the model train. To the immediate right of the computer behind the train set is the MD100 LIM. Following the trade fair, for several years, we used the train set for programming exercises in Erlang courses, until the points wore out through excessive use. While the control software was fault-tolerant, the hardware was far less reliable and we were plagued with small mechanical problems.

4.10 The collapse of AXE-N

In December 1995, a large project at Ellemtel, called AXE-N, collapsed. This was the single most important event in the history of Erlang. Without the collapse of AXE-N, Erlang would have still remained a Lab experiment and the effort to turn it into a commercial-quality product would not have happened. The difference is the many thousands of hours of work that must be done to produce high-quality documentation and to produce and test extensive libraries. AXE-N was a project aimed at developing a new generation of switching products ultimately to replace the AXE10 system. The AXE-N project had developed a new hardware platform and system software that was developed in C++.

Following a series of crisis meetings the project was reorganised and re-started. This time the programming language would be

¹⁵ MD110 LIM (Line Interface Module).

Erlang and hardware from the AXE-N project was salvaged to start the production of a new ATM¹⁶ switch, to be called the AXD. This new project was to be the largest-ever Erlang project so far, with over 60 Erlang programmers. At the start of the AXD project, the entire Erlang system was the responsibility of half a dozen people in the Lab. This number was viewed as inadequate to support the needs of a large development project and so plans were immediately enacted to build a product unit, called OTP, to officially support the Erlang system. At this time all external marketing of Erlang was stopped, since all available “resources” should now focus on internal product development.

OTP stands for the “Open Telecom Platform” and is both the name of the Erlang software distribution and the name of an Ericsson product unit, which can be somewhat confusing. The OTP unit started in the Lab but in 1997 was formed as a new product unit outside the Lab. Since 1997, the OTP unit has been responsible for the distribution of Erlang.

4.11 BOS – OTP and behaviors

Alongside the Erlang language development, the question of libraries and middleware has always been important. Erlang is just a programming language, and to build a complete system something more than just a programming language is needed. To build any significant body of software you need not only a programming language but a significant set of libraries and some kind of operating system to run everything on. You also need a philosophy of programming—since you cannot build a large body of software in an ad-hoc manner without some guiding principles.

The collective name for Erlang, all the Erlang libraries, the Erlang run-time system and descriptions of the Erlang way of doing things is the OTP system. The OTP system contains:

- Libraries of Erlang code.
- Design patterns for building common applications.
- Documentation.
- Courses.
- How to’s.

The libraries are organised and described in a conventional manner. They also have pretty conventional semantics. One reviewer of this paper asked how we integrated side effects with our language, for example what happens if a open file handle is sent in a message to two different processes. The answer is that side effects like this are allowed. Erlang is not a strict side-effect-free functional language but a concurrent language where what happens inside a process is described by a simple functional language. If two different processes receive a Pid representing a file, both are free to send messages to the file process in any way they like. It is up to the logic of the application to prevent this from happening.

Some processes are programmed so that they only accept messages from a particular process (which we call the owning process). In this case problems due to sharing a reference can be avoided, but code libraries do not necessarily have to follow such a convention.

In practice this type of problem rarely presents problems. Most programmers are aware of the problems that would arise from shared access to a resource and therefore use mnesia transactions or functions in the OTP libraries if they need shared access to a resource.

What is more interesting is the set of design patterns included in the OTP system. These design patterns (called behaviors) are the result of many years’ experience in building fault-tolerant systems. They are typically used to build things like client-server mod-

```
-module(server).
-export([start/2, call/2, change_code/2]).

start(Fun, Data) ->
    spawn(fun() -> server(Fun, Data) end).

call(Server, Args) ->
    rpc(Server, {query, Args}).

change_code(Server, NewFunction) ->
    rpc(Server, {new_code, NewFunction}).

rpc(Server, Query) ->
    Server ! {self(), Query},
    receive
        {Server, Reply} -> Reply
    end.

server(Fun, Data) ->
    receive
        {From, {query, Query}} ->
            {Reply, NewData} = Fun(Query, Data),
            From ! {self(), Reply},
            server(Fun, NewData);
        {from, {swap_code, NewFunction}} ->
            From ! {self(), ack},
            server(Data, NewFunction)
    end.
```

Figure 9. A generic client-server model with hot-code replacement.

els, event-handling systems etc. Behaviors in Erlang can be thought of as parameterizable higher-order parallel processes. They represent an extension of conventional higher-order functions (like map, fold etc) into a concurrent domain.

The design of the OTP behaviors was heavily influenced by two earlier efforts. The first was a system called BOS.¹⁷ BOS was an application operating system written at Bollmora in Erlang specifically for the Mobility Server project. The BOS had solved a number of problems in a generic manner, in particular how to build a generic server and how to build a generic kind of error supervisor. Most of this had been done by Peter Höglfeldt. The second source of inspiration was a generic server implemented by Klacke.

When the OTP project started, I was responsible for the overall technology in the project and for developing a new set of behaviors that could be used for building fault-tolerant systems. This in turn led to the development of a dozen or so behaviors, all of which simplify the process of building a fault-tolerant system. The behaviors abstract out things like failure so that client-server models can be written using simple functional code in such a manner that the programmer need only be concerned with the functionality of the server and not what will happen in the event of failure or distribution. This part of the problem is handled by the generic component of the behavior.

The other two people who were heavily involved in the development of the behaviors were Martin Björklund and Magnus Fröberg. Unfortunately space limitations preclude a more extensive treatment of behaviors. Figure 9 has a greatly simplified sketch of a client-server behavior. Note that this model provides the normal functionality of a client-server and the ability to hot-swap the code. The rationale behind behaviors and a complete set of examples can be found in [7].

¹⁶ Asynchronous Transfer Mode.

¹⁷ Basic Operating System.

4.12 More language changes

During the period 1989 to 1998, i.e. from the existence of a stable JAM-based system and up to the release of Open Source Erlang, a number of changes crept into the language. These can be categorised as major or minor changes. In this context, minor means that the change could be categorised as a simple incremental improvement to the language. Minor changes suggest themselves all the time and are gradually added to the language without much discussion. Minor changes make the programmer's life a lot easier, but do not affect how we think about the programming process itself. They are often derived from ideas in conventional programming languages that are assimilated into Erlang.

The minor changes which were added included:

- Records.
- Macros.
- Include files.
- Infix notation for list append and subtract ("++" and "-").

The major changes are covered in the following sections.

4.13 Influence from functional programming

By now the influence of functional programming on Erlang was clear. What started as the addition of concurrency to a logic language ended with us removing virtually all traces of Prolog from the language and adding many well-known features from functional languages.

Higher-order functions and list comprehensions were added to the language. The only remaining signs of the Prolog heritage lie in the syntax for atoms and variables, the scoping rules for variables and the dynamic type system.

4.14 Binaries and the bit syntax

A binary is a chunk of untyped data, a simple memory buffer with no internal structure. Binaries are essential for storing untyped data such as the contents of a file or of a data packet in a data communication protocol. Typical operations on a binary often include dividing it into two, according to some criteria, or combining several small binaries to form a larger binary. Often binaries are passed unmodified between processes and are used to carry input/output data. Handling binary data efficiently is an extremely difficult problem and one which Klacke and Tony Rogvall spent several years implementing.

Internally binaries are represented in several different ways, depending upon how and when they were created and what has happened to them since their creation. Sending messages containing binaries between two processes in the same node does not involve any copying of the binaries, since binaries are kept in a separate reference-counted storage area that is not part of the stack and heap memory which each process has.

The bit syntax [27] is one of those unplanned things that was added in response to a common programming problem. Klacke and Tony had spent a long time implementing various low-level communication protocols in Erlang. In so doing, the problem of packing and unpacking bit fields in binary data occurred over and over again. To unpack or pack such a data structure, Tony and Klacke invented the bit syntax and enhanced Erlang pattern matching to express patterns over bit fields.

As an example, suppose we have a sixteen-bit data structure representing a ten-bit counter, three one-bit flags and a three-bit status indicator. The Erlang code to unpack such a structure is:

```
<<N:10,Flag1:1,Flag2:1,Flag3:1,Status:3>> = B
```

This is one of those simple ideas which after you have seen it makes you wonder how any language could be without it. Using

the bit syntax yields highly optimised code that is extremely easy to write and fits beautifully with the way most low-level protocols are specified.

4.15 Mnesia ETS tables and databases

In developing large-scale telecommunications applications it soon became apparent that the "pure" approach of storing data could not cope with the demands of a large project and that some kind of real-time database was needed. This realization resulted in a DBMS called Mnesia¹⁸ [24, 25, 21]. This work was started by Klacke but soon involved Hans Nilsson, Törbjörn Törnvist, Håkan Matsson and Tony Rogvall. Mnesia had both high- and low-level components. At the highest level of abstract was a new query language called Mnemosyne (developed by Hans Nilsson) and at the lowest level were a set of primitives in Erlang with which Mnesia could be written. Mnesia satisfied the following requirements (from [21]):

1. Fast Key/Value lookup.
2. Complicated non real-time queries, mainly for operation and maintenance.
3. Distributed data due to distributed applications.
4. High fault tolerance.
5. Dynamic reconfiguration.
6. Complex objects.

In order to implement Mnesia in Erlang, one additional Erlang module had to be developed. This was the Erlang module `ets`, short for Erlang term storage. `Ets` provided low-level destructive term storage based on extensible hash tables. Although `ets` looks as if it had been implemented in Erlang (i.e. it is an Erlang module), most of its implementation is contained in the Erlang virtual machine implementation.

4.16 High-performance Erlang

The HiPE (High-Performance Erlang) project is a research project at the Department of Information Technology at the University of Uppsala. The HiPE team have concentrated on efficient implementation of Erlang and type checking systems for Erlang. This project runs in close collaboration with members of the OTP group. Since 2001, the HiPE native code compiler has been an integral part of the Open Source Erlang distribution.

4.17 Type inference of Erlang programs

Erlang started life as a Prolog interpreter and has always had a dynamic type system, and for a long time various heroic attempts have been made to add a type system to Erlang. Adding a type system to Erlang seems at first a moderately difficult endeavour, which on reflection becomes impossibly difficult.

The first attempt at a type system was due to an initiative taken by Phil Wadler. One day Phil phoned me up and announced that a) Erlang needed a type system, b) he had written a small prototype of a type system and c) he had a one year's sabbatical and was going to write a type system for Erlang and "were we interested?" Answer — "Yes."

Phil Wadler and Simon Marlow worked on a type system for over a year and the results were published in [20]. The results of the project were somewhat disappointing. To start with, only a subset of the language was type-checkable, the major omission being the lack of process types and of type checking inter-process messages. Although their type system was never put into production,

¹⁸The original name was Amnesia until a senior Ericsson manager noticed the name. "It can't possibly be called Amnesia," he said, "the name must be changed" — and so we dropped the "a."

it did result in a notation for types which is still in use today for informally annotating types.

Several other projects to type check Erlang also failed to produce results that could be put into production. It was not until the advent of the Dialyzer¹⁹ [18] that realistic type analysis of Erlang programs became possible. The Dialyzer came about as a side-effect of the HiPE project mentioned earlier. In order to efficiently compile Erlang, a type analysis of Erlang programs is performed. If one has precise information about the type of a function, specialised code can be emitted to compile that function, otherwise generic code is produced. The HiPE team took the view that complete information about all the types of all the variables in all the statements of an Erlang program was unnecessary and that any definite statements about types, even of a very small subsection of a program, provided useful information that could guide the compiler into generating more efficient code.

The Dialyzer does not attempt to infer all types in a program, but any types it does infer are guaranteed to be correct, and in particular any type errors it finds are guaranteed to be errors. The Dialyzer is now regularly used to check large amounts of production code.

5. Part IV: 1998 – 2001. Puberty problems — the turbulent years

1998 was an exciting year in which the following events occurred:

- The first demo of GPRS²⁰ developed in Erlang was demonstrated at the GSM World Congress in February and at CeBIT in March.
- In February, Erlang was banned inside Ericsson Radio Systems.
- In March, the AXD301 was announced. This was possibly the largest ever program in a functional language.
- In December, Open Source Erlang was released.
- In December, most of the group that created Erlang resigned from Ericsson and started a new company called *Bluetail AB*.

5.1 Projects Succeed

In 1998, the first prototype of a GPRS system was demonstrated and the Ericsson AXD301 was announced. Both these systems were written in a mixture of languages, but the main language for control in both systems was Erlang.

The largest ever system built in Erlang was the AXD301. At the time of writing, this system has 2.6 millions lines of Erlang code. The success of this project demonstrates that Erlang is suitable for large-scale industrial software projects. Not only is the system large in terms of code volume, it is also highly reliable and runs in realtime. Code changes in the system have to be performed without stopping the system. In the space available it is difficult to describe this system adequately so I shall only give a brief description of some of the characteristics.

The AXD301 is written using distributed Erlang. It runs on a cluster using pairs of processors and is scalable up to 16 pairs of processors. Each pair is “self contained,” which means that if one processor in the pair fails, the other takes over. The take-over mechanisms and call control are all programmed in Erlang. Configuration data and call control data are stored in a Mnesia database that can be accessed from any node and is replicated on several nodes. Individual nodes can be taken out of service for repair, and additional nodes can be added without interrupting services.

¹⁹ DIscrEpancy AnaLYZer of ERlang programs.

²⁰ General Packet Radio Service.

The software for the system is programmed using the behaviors from the OTP libraries. At the highest level of abstraction are a number of so-called “supervision trees”—the job of a node in the supervision tree is to monitor its children and restart them in the event of failure. The nodes in a decision tree are either supervision trees or primitive OTP behaviors. The primitive behaviors are used to model client-servers, event-loggers and finite-state machines. In the analysis of the AXD reported in [7], the AXD used 20 supervision trees, 122 client-server models, 36 event loggers and 10 finite-state machines.

All of this was programmed by a team of 60 programmers. The vast majority of these programmers had an industrial background and no prior knowledge of functional or concurrent programming languages. Most of them were taught Erlang by the author and his colleagues. During this project the OTP group actively supported the project and provided tool support where necessary. Many in-house tools were developed to support the project. Examples include an ASN.1 compiler and in-built support for SNMP in Mnesia.

The OTP behaviors themselves were designed to be used by large groups of programmers. The idea was that there should be one way to program a client-server and that all programmers who needed to implement a client server would write plug-in code that slotted into a generic client-server framework. The generic server framework provided code for all the tricky parts of a client-server, taking care of things like code change, name registration, debugging, etc. When you write a client-server using the OTP behaviors you need only write simple sequential functions: all the concurrency is hidden inside the behavior.

The intention in the AXD was to write the code in as clear a manner as possible and to mirror the specifications exactly. This turned out to be impossible for the call control since we ran into memory problems. Each call needed six processes and processing hundreds of thousands of calls proved impossible. The solution to this was to use six processes per call only when creating and destroying a call. Once a call had been established, all the processes responsible for the call were killed and data describing the call was inserted into the real-time database. If anything happened to the call, the database entry was retrieved and the call control processes recreated.

The AXD301 [8] was a spectacular success. As of 2001, it had 1.13 million lines of Erlang code contained in 2248 modules [7]. If we conservatively estimate that one line of Erlang would correspond to say five lines of C, this corresponds to a C system with over six million lines of code.

As regards reliability, the AXD301 has an observed nine-nines reliability [7]—and a four-fold increase in productivity was observed for the development process [31].

5.2 Erlang is banned

Just when we thought everything was going well, in 1998, Erlang was banned within Ericsson Radio AB (ERA) for new product development. This ban was the second most significant event in the history of Erlang: It led indirectly to Open Source Erlang and was the main reason why Erlang started spreading outside Ericsson. The reason given for the ban was as follows:

The selection of an implementation language implies a more long-term commitment than the selection of a processor and OS, due to the longer life cycle of implemented products. Use of a proprietary language implies a continued effort to maintain and further develop the support and the development environment. It further implies that we cannot easily benefit from, and find synergy with, the evolution following the large scale deployment of globally used languages. [26] quoted in [12].

In addition, projects that were already using Erlang were allowed to continue but had to make a plan as to how dependence upon Erlang could be eliminated. Although the ban was only within ERA, the damage was done. The ban was supported by the Ericsson technical directorate and flying the Erlang flag was thereafter not favored by middle management.

5.3 Open Source Erlang

Following the Erlang ban, interest shifted to the use of Erlang outside Ericsson.

For some time, we had been distributing Erlang to interested parties outside Ericsson, although in the form of a free evaluation system subject to a non-disclosure agreement. By 1998, about 40 evaluation systems had been distributed to external users and by now the idea of releasing Erlang subject to an open source license was formed. Recall that at the start of the Erlang era, in 1986, “open source” was unheard of, so in 1986 everything we did was secret. By the end of the era, a significant proportion of the software industry was freely distributing what they would have tried to sell ten years earlier—as was the case with Erlang.

In 1998, Jane Walerud started working with us. Jane had the job of marketing Erlang to external users but soon came to the conclusion that this was not possible. There was by now so much free software available that nobody was interested in buying Erlang. We agreed with Jane that selling Erlang was not viable and that we would try to get approval to release Erlang subject to an open source license. Jane started lobbying the management committee that was responsible for Erlang development to persuade it to approve an open source release. The principal objection to releasing Erlang as Open Source was concerned with patents, but eventually approval was obtained to release the system subject to a patent review. On 2 December 1998, Open Source Erlang was announced.

5.4 Bluetail formed

Shortly after the open source release, the majority of the original Erlang development team resigned from Ericsson and started a new company called Bluetail AB with Jane as the chief executive. In retrospect the Erlang ban had the opposite effect and stimulated the long-term growth of Erlang. The ban led indirectly to Open Source Erlang and to the formation of Bluetail. Bluetail led in its turn to the introduction of Erlang into Nortel Networks and to the formation of a small number of Erlang companies in the Stockholm region.

When we formed Bluetail, our first decision was to use Erlang as a language for product development. We were not interested in further developing the language nor in selling any services related to the language. Erlang gave us a commercial advantage and we reasoned that by using Erlang we could develop products far faster than companies using conventional techniques. This intuition proved to be correct. Since we had spent the last ten years designing and building fault-tolerant telecoms devices, we turned our attention to Internet devices, and our first product was a fault-tolerant e-mail server called the mail robustifier.

Architecturally this device has all the characteristics of a switching system: large numbers of connections, fault-tolerant service, ability to remove and add nodes with no loss of service. Given that the Bluetail system was programmed by most of the people who had designed and implemented the Erlang and OTP systems, the project was rapidly completed and had sold its first system within six months of the formation of the company. This was one of the first products built using the OTP technology for a non-telecoms application.

5.5 The IT boom – the collapse and beyond

From 1998 to 2000 there were few significant changes to Erlang. The language was stable and any changes that did occur were

under the surface and not visible to external users. The HiPE team produced faster and faster native code compilers and the Erlang run-time system was subject to continual improvement and revision in the capable hands of the OTP group.

Things went well for Bluetail and in 2000, the company was acquired by Alteon Web systems and six days later Alteon was acquired by Nortel Networks. Jane Walerud was voted Swedish IT person of the year. Thus is was that Erlang came to Nortel Networks. The euphoric period following the Bluetail acquisition was short-lived. About six months after the purchase, the IT crash came and Nortel Networks fired about half of the original Bluetail gang. The remainder continued with product development within Nortel.

6. Part V: 2002 – 2005. Coming of age

By 2002, the IT boom was over and things had begun to calm down again. I had moved to SICS²¹ and had started thinking about Erlang again. In 2002, I was fortunate in being asked to hold the opening session at the second Lightweight Languages Symposium (held at MIT).

6.1 Concurrency oriented programming and the future

In preparing my talk for LL2 I tried to think of a way of explaining what we had been doing with Erlang for the last 15 years. In so doing, I coined the phrase “concurrency oriented programming”—at the time I was thinking of an analogy with object oriented programming. As regards OO programming I held the view that:

- An OO language is characterised by a vague set of rules.
- Nobody agrees as to what these rules are.
- Everybody knows an OO language when they see one.

Despite the fact that exactly what constitutes an OO language varies from language to language, there is a broad understanding of the principles of OO programming and software development. OO software development is based first upon the identification of a set of objects and thereafter by the sets of functions that manipulate these objects.

The central notion in concurrency oriented programming (COP) is to base the design on the concurrency patterns inherent in the problem. For modelling and programming real-world objects this approach has many advantages—to start with, things in the real world happen concurrently. Trying to model real-world activities without concurrency is extremely difficult.

The main ideas in COP are:

- Systems are built from processes.
- Process share nothing.
- Processes interact by asynchronous message passing.
- Processes are isolated.

By these criteria both PLEX and Erlang can be described as concurrency oriented languages.

This is then what we have been doing all along. The original languages started as a sequential language to which I added processes, but the goal of this was to produce lightweight concurrency with fast message passing.

The explanations of what Erlang was have changed with time:

1. 1986 – Erlang is a declarative language with added concurrency.
2. 1995 – Erlang is a functional language with added concurrency.

²¹Swedish Institute of Computer Science.

3. 2005 – Erlang is a concurrent language consisting of communicating components where the components are written in a functional language. Interestingly, this mirrors earlier work in Eri-Pascal where components were written in Pascal.

Now 3) is a much better match to reality than ever 1) or 2) was. Although the functional community was always happy to point to Erlang as a good example of a functional language, the status of Erlang as a fully fledged member of the functional family is dubious. Erlang programs are not referentially transparent and there is no system for static type analysis of Erlang programs. Nor is it a relational language. Sequential Erlang has a pure functional subset, but nobody can force the programmer to use this subset; indeed, there are often good reasons for not using it.

Today we emphasize the concurrency. An Erlang system can be thought of as a communicating network of black boxes. If two black boxes obey the principle of observational equivalence, then for all practical purposes they are equivalent. From this point of view, the language used inside the black box is totally irrelevant. It might be a functional language or a relational language or an imperative language—in understanding the system this is an irrelevant detail.

In the Erlang case, the language inside the black box just happens to be a small and rather easy to use functional language, which is more or less a historical accident caused by the implementation techniques used.

If the language inside the black boxes is of secondary importance, then what is of primary importance? I suspect that the important factor is the interconnection paths between the black boxes and the protocols observed on the channels between the black boxes.

As for the future development of Erlang, I can only speculate. A fruitful area of research must be to formalise the interprocess protocols that are used and observed. This can be done using synchronous calculi, such as CSP, but I am more attracted to the idea of protocol checking, subject to an agreed contract. A system such as UBF [6] allows components to exchange messages according to an agreed contract. The contract is checked dynamically, though I suspect that an approach similar to that used in the Dialyzer could be used to remove some of the checks.

I also hope that the Erlang concurrency model and some of the implementation tricks²² will find their way into other programming languages. I also suspect that the advent of true parallel CPU cores will make programming parallel systems using conventional mutexes and shared data structures almost impossibly difficult, and that the pure message-passing systems will become the dominant way to program parallel systems.

I find that I am not alone in this belief. Paul Morrison [23] wrote a book in 1992 suggesting that flow-based programming was the ideal way to construct software systems. In his system, which in many ways is very similar to Erlang, interprocess pipes between processes are first-class objects with infinite storage capacity. The pipes can be turned on and off and the ends connected to different processes. This view of the world concentrates on the flow of data between processes and is much more reminiscent of programming in the process control industry than of conventional algorithmic programming. The stress is on data and how it flows through the system.

6.2 Erlang in recent times

In the aftermath of the IT boom, several small companies formed during the boom have survived, and Erlang has successfully re-rooted itself outside Ericsson. The ban at Ericsson has not succeeded in completely killing the language, but it has limited its growth into new product areas.

²²Like the bit pattern matching syntax.

The plans within Ericsson to wean existing projects off Erlang did not materialise and Erlang is slowly winning ground due to a form of software Darwinism. Erlang projects are being delivered on time and within budget, and the managers of the Erlang projects are reluctant to make any changes to functioning and tested software.

The usual survival strategy within Ericsson during this time period was to call Erlang something else. Erlang had been banned but OTP hadn't. So for a while no new projects using Erlang were started, but it was OK to use OTP. Then questions about OTP were asked: "Isn't OTP just a load of Erlang libraries?"—and so it became "Engine," and so on.

After 2002 some of the surviving Bluetail members who moved to Nortel left and started a number of 2nd-generation companies, including Tail-F, Kreditor and Synapse. All are based in the Stockholm region and are thriving.

Outside Sweden the spread of Erlang has been equally exciting. In the UK, an ex-student of mine started Erlang Consulting, which hires out Erlang consultants to industry. In France, Process-one makes web stress-testing equipment and instant-messaging solutions. In South Africa, Erlang Financial Systems makes banking software. All these external developments were spontaneous. Interested users had discovered Erlang, installed the open-source release and started programming. Most of this community is held together by the Erlang mailing list, which has thousands of members and is very active. There is a yearly conference in Stockholm that is always well attended.

Recently, Erlang servers have begun to find their way into high-volume Internet applications. Jabber.org has adopted the ejabberd instant messaging server, which is written in Erlang and supported by Process-one.

Perhaps the most exciting modern development is Erlang for multicore CPUs. In August 2006 the OTP group released Erlang for an SMP. In most other programming communities, the challenge of the multicore CPU is to answer the question, "How can I parallelize my program?" Erlang programmers do not ask such questions; their programs are already parallel. They ask other questions, like "How can I increase the parallelism in an already parallel program?" or "How can I find the bottlenecks in my parallel program?" but the problem of parallelization has already been solved.

The "share nothing pure message passing" decisions we took in the 1980s produce code which runs beautifully on a multicore CPU. Most of our programs just go faster when we run them on a multicore CPU. In an attempt to further increase parallelism in an already parallel program, I recently wrote a parallel version of map (pmap) that maps a function over a list in parallel. Running this on a Sun Fire T2000 Server, an eight core CPU with four threads per core, made my program go 18 times faster.

6.3 Mistakes made and lessons learnt

If we are not to make the same mistakes over and over again then we must learn from history. Since this is the history of Erlang, we can ask, "What are the lessons learnt? the mistakes made? what was good? what was bad?" Here I will discuss some of what I believe are the generic lessons to be learned from our experience in developing Erlang, then I will talk about some of the specific mistakes.

First the generic lessons:

The Erlang development was driven by the prototype

Erlang started as a prototype and during the early years the development was driven by the prototype; the language grew slowly in response to what we and the users wanted. This is how we worked.

First we wrote the code, then we wrote the documentation. Often the users would point out that the code did not do what the documentation said. At this phase in the development we told

them, “If the code and the documentation disagree then the code is correct and the documentation wrong.” We added new things to the language and improved the code and when things had stabilized, we updated the documentation to agree with the code.

After a couple of years of this way of working, we had a pretty good user’s manual [3]. At this point we changed our way of working and said that from now on the manuals would only describe what the language was supposed to do and if the implementation did something else then it was a bug and should be reported to us. Once again, situations would be found when the code and the documentation did not agree, but now it was the code that was wrong and not the documentation.

In retrospect this seems to be the right way of going about things. In the early days it would have been totally impossible to write a sensible specification of the language. If we had sat down and carefully thought out what we had wanted to do before doing it, we would have got most of the details wrong and would have had to throw our specifications away.

In the early days of a project, it is extremely difficult to write a specification of what the code is supposed to do. The idea that you can specify something without having the knowledge to implement it is a dangerous approximation to the truth. Language specifications performed without knowledge of how the implementation is to be performed are often disastrously bad. The way we worked here appears to be optimal. In the beginning we let our experiments guide our progress. Then, when we knew what we were doing, we could attempt to write a specification.

Concurrent processes are easy to compose

Although Erlang started as a language for programming switches, we soon realized that it was ideal for programming many general-purpose applications, in particular, any application that interacted with the real world. The pure message-passing paradigm makes connecting Erlang processes together extremely easy, as is interfacing with external applications. Erlang views the world as communicating black boxes, exchanging streams of message that obey defined protocols. This make it easy to isolate and compose components. Connecting Erlang processes together is rather like Unix shell programming. In the Unix shell we merely pipe the output of one program into the input of another. This is exactly how we connect Erlang processes together: we connect the output of one process to the input of another. In a sense this is even easier than connecting Unix processes with a pipe, as in the Erlang case the messages are Erlang terms that can contain arbitrary complex data structures requiring no parsing. In distributed Erlang the output of one program can be sent to the input of another process on another machine, just as easily as if it had been on the same machine. This greatly simplifies the code.

Programmers were heavily biased by what the language does and not by what it should do

Erlang programmers often seem to be unduly influenced by the properties of the current implementation. Throughout the development of Erlang we have found that programming styles reflected the characteristics of the implementation. So, for example, when the implementation limited the maximum number of processes to a few tens of thousands of processes, programmers were overly conservative in their use of processes. Another example can be found in how programmers use atoms. The current Erlang implementation places restrictions on the maximum number of atoms allowed in the system. This is a hard limit defined when the system is built. The atom table is also not subject to garbage collection. This has resulted in lengthy discussion on the Erlang mailing lists and a reluctance to use dynamically recreated atoms in application programs. From the implementor’s point of view, it would be better to encourage programmers to use atoms when appropriate and then fix the implementation when it was not appropriate.

In extreme cases, programmers have carefully measured the most efficient way to write a particular piece of code and then adopted this programming style for writing large volumes of code. A better approach would be to try to write the code as beautifully and clearly as possible and then, if the code is not fast enough, ask for the implementor’s help in speeding up the implementation.

People are not convinced by theory, only by practice

We have often said that things could be done (that they were theoretically possible) but did not actually do them. Often our estimates of how quickly we could do something were a lot shorter than was generally believed possible. This created a kind of credibility gap where we did not implement something because we thought it was really easy, and the management thought we did not know what we were talking about because we had not actually implemented something. In fact, both parties were probably incorrect; we often underestimated the difficulty of an implementation and the management overestimated the difficulty.

The language was not planned for change from the beginning

We never really imagined that the language itself would evolve and spread outside the Lab. So there are no provisions for evolving the syntax of the language itself. There are no introspection facilities so that code can describe itself in terms of its interfaces and versions, etc.

The language has fixed limits and boundaries

Just about every decision to use a fixed size data structure was wrong. Originally Pids (process identifiers) were 32-bit stack objects—this was done for “efficiency reasons.” Eventually we couldn’t fit everything we wanted to describe a process into 32 bits, so we moved to larger heap objects and pointers. References were supposed to be globally unique but we knew they were not. There was a very small possibility that two identical references might be generated, which, of course, happened.

Now for the specific lessons:

There are still a number of areas where Erlang should be improved. Here is a brief list:

We should have atom GC Erlang does not garbage collect atoms.

This means that some programs that should be written using atoms are forced to use lists or binaries (because the atom table might overflow).

We should have better ways interfacing foreign code Interfacing non-Erlang code to Erlang code is difficult, because the foreign code is not linked to the Erlang code for safety reasons. A better way of doing this would be to run the foreign code in distributed Erlang nodes, and allow foreign language code to be linked into these “unsafe” nodes.

We should improve the isolation between processes Process iso-

lation is not perfect. One process can essentially perform a “denial of service attack” on another process by flooding it with messages or by going into an infinite loop to steal CPU cycles from the other processes. We need safety mechanisms to prevent this from happening.

Safe Erlang Security in distributed Erlang is “all or nothing,” meaning that, once authenticated, a distributed Erlang node can perform any operation on any other node in the system. We need a security module that allows distributed nodes to process remote code with varying degrees of trust.

We need notations to specify protocols and systems Protocols themselves are not entities in Erlang, they are not named and they can be inferred only by reading the code in a program. We need more formal ways of specifying protocols and run-time methods for ensuring that the agents involved in implementing a protocol actually obey that protocol.

Code should be first class Functions in Erlang are first-class, but the modules themselves are not first-class Erlang objects. Modules should also be first-class objects: we should allow multiple versions of module code²³ and use the garbage collector to remove old code that can no longer be evaluated.

6.4 Finally

It is perhaps interesting to note that the two most significant factors that led to the spread of Erlang were:

- The collapse of the AXE-N project.
- The Erlang ban.

Both of these factors were outside our control and were unplanned. These factors were far more significant than all the things we did plan for and were within our control. We were fortuitously able to take advantage of the collapse of the AXE-N project by rushing in when the project failed. That we were able to do so was more a matter of luck than planning. Had the collapse occurred at a different site then this would not have happened. We were able to step in only because the collapse of the project happened in the building where we worked so we knew all about it.

Eventually Ericsson did the right thing (using the right technology for the job) for the wrong reasons (competing technologies failed). One day I hope they will do the right things for the right reasons.

²³ Erlang allows two versions of the same module to exist at any one time, this is to allow dynamic code-upgrade operations.

A. Change log of erlang.pro

24 March 1988 to 14 December 1988

```
/* $HOME/erlang.pro
 * Copyright (c) 1988 Ericsson Telecom
 * Author: Joe Armstrong
 * Creation Date: 1988-03-24
 * Purpose:
 *   main reduction engine
 *
 *
 * Revision History:
 * 88-03-24      Started work on multi processor version of erlang
 * 88-03-28      First version completed (Without timeouts)
 * 88-03-29      Correct small errors
 * 88-03-29      Changed 'receive' to make it return the pair
 *                msg(From,Mess)
 * 88-03-29      Generate error message when out of goals
 *                i.e. program doesn't end with terminate
 * 88-03-29      added trace(on), trace(off) facilities
 * 88-03-29      Removed Var :_ {....} , this can be achieved
 *                with {...}
 * 88-05-27      Changed name of file to erlang.pro
 *                First major revision started - main changes
 *                Complete change from process to channel
 *                based communication here we (virtually) throw away all the
 *                old stuff and make a bloody great data base
 * 88-05-31      The above statements were incorrect much better
 *                to go back to the PROPER way of doing things
 *                long live difference lists
 * 88-06-02      Reds on run([et5]) = 245
 *                Changing the representation to separate the
 *                environment and the process - should improve things
 *                It did .... reds = 283 - and the program is nicer!
 * 88-06-08      All pipe stuff working (pipes.pro)
 *                added code so that undefined functions can return
 *                values
 * 88-06-10      moved all stuff to /dunder/sys3
 *                decided to remove all pipes !!!!!! why?
 *                mussy semantics - difficult to implement
 *                This version now 3.01
 *                Changes case, and reduce Rhs's after =>
 *                to allow single elements not in list brackets
 * 88-06-13      added link(Proc), unlink(Proc) to control
 *                error recovery.
 *                a processes that executes error-exit will send
 *                a kill signal to all currently linked processes
 *                the receiving processes will be killed and will
 *                send kill's to all their linked processes etc.
 * 88-06-14      corrected small error in kill processing
 *                changed name of spy communications(on/off)
 *                to trace comms(on/off)
 * 88-06-16      added load(File) as an erlang command
 *                added function new_ref - retruns
 *                a unique reference
 * 88-06-22      added structure parameter io_env
 *                to hold the io_environment for LIM communication
 *                changes required to add communication with lim ...
 *                no change to send or receive the hw will just appear
 *                as a process name (eg send(tsu(16),...))
 *                note have to do link(...) before doing a send
 *                change to top scheduler(msg(From,To,...))
 *                such that message is sent to Hw if To represents Hw
 *                following prims have been added to prims.tel
 *                link _hw(Hw,Proc) - send all msgs from Hw to Proc
```

```

*
* unlink _hw(Hw) - stop it
* unlink _all hw -- used when initialising
* added new primitive
* init hw causes lim to be initialised
* load-magic ... links the c stuff
* simulate(onloff) -- really send to HW
* 88-06-27 Bug is bind _var ...
* foo(A,B) := zap, ...
* failed when zap returned foo(aa,bb)
* 88-07-04 port to quintus --- change $$ variables
* change the internal form of erlang clauses
* i.e. remove clause(...)
* 88-07-07 changed order in receive so that first entry is
in queu is pulled out if at all possible
* 88-07-08 exit(X) X <> normal causes trace printout
* 88-09-14 ported to SICSTUS -
changed load to eload to avoid name clash with
SICSTUS
* 88-10-18 changed the return variable strategy.
don't have to say void := Func to throw away the
return value of a function
* 88-10-18 If we hit a variable on the top of the reduction
stack, then the last called function did not return
a value, we bind the variable to undefined
(this means that write,nl,... etc) now all return
undefined unless explicitly overridden
<<< does case etc return values correctly ?>
[[[ I hope so ]]]
* 88-10-18 send just sends doesn't check for a link
* 88-10-19 reworked the code for link and unlink
multiple link or unlink doesn't bugger things
make link by-directional. This is done by linking
locally and sending an link/unlink messages to
the other side
* 88-10-19 add command trap exit(Arg) .. Arg = yes I no
Implies extra parameter in facts/4
* 88-10-19 Changed the semantics of exit as follows:
error_exit is removed
exit(Why) has the following semantics
when exit(Anything) is encountered an exit(Why)
message is sent to all linked processes the action
taken at ther receiving end is as follows:
1) if trap exit(no) & exit(normal)
message is scrapped
2) if trap exit(no) & exit(X) & X <> normal
exit(continue) is send to all linked processes
EXCEPT the originating process
3) if trap_exit(yes) then the exit message
is queued for reception with a
receive([
    exit(From,Why) =>
    ...
    statement
* 88-10-19 send_sys(Term) implemented .. used for faking
up a internal message
* 88-10-24 can now do run(Mod:Goal) ..
* 88-10-25 fixed spawn(Mod:Goal,Pri) to build function name
at run time
* 88-10-27 All flags changes to yes I no
i.e. no more on,off true, false etc.
* 88-11-03 help command moved to top.pl
* 88-11-03 changed top scheduler to carry on
reducing until an empty queu is reached and then stop
changed 1111 to 'sys$$call
* 88-11-08 added lots of primitives (read the code!)

```

```
* 88-11-08 removed simulate(on) ... etc.  
* must be done from the top loop  
* 88-11-17 added link/2, unlink/2  
* 88-11-17 added structure manipulating primitives  
* atom_2_list(Atom),list_2_atom(List),  
* struct_name(Struct), struct_args(AStruct),  
* struct_arity(Struct), make_struct(Name,Args)  
* get_arg(Argno,Struct), set_arg(Argno,Struct,Value)  
* 88-11-17 run out of goals simulates exit(normal)  
* 88-11-17 and timeout messages  
* 88-12-14 added two extra parameters to facts  
* Save-messages(yeslno) and alias  
*/
```

B. Erlang examples

B.1 Sequential Erlang examples

Factorial

All code is contained in modules. Only exported functions can be called from outside the modules.

Function clauses are selected by pattern matching.

```
-module(math).
-export([fac/1]).

fac(N) when N > 0 -> N * fac(N-1);
fac(0) -> 1.
```

We can run this in the Erlang shell.²⁴

```
> math:fac(25).
15511210043330985984000000
```

Binary Trees

Searching in a binary tree. Nodes in the tree are either nil or {Key, Val, S, G} where S is a tree of all nodes less than Key and G is a tree of all nodes greater than Key.

Variables in Erlang start with an uppercase letter. Atoms start with a lowercase letter.

```
lookup(Key, {Key, Val, _, _}) ->
    {ok, Val};
lookup(Key, {Key1, Val, S, G}) when Key < Key1 ->
    lookup(Key, S);
lookup(Key, {Key1, Val, S, G}) ->
    lookup(Key, G);
lookup(Key, nil) ->
    not_found.
```

Append

Lists are written [H|T]²⁵ where H is any Erlang term and T is a list. [X1, X2, ..., Xn] is shorthand for [X1 | [X2 | ... | [Xn | []]]].

```
append([H|T], L) -> [H|append(T, L)];
append([], L) -> L.
```

Sort

This makes use of list comprehensions:

```
sort([Pivot|T]) ->
    sort([X||X <- T, X < Pivot]) ++
    [Pivot] ++
    sort([X||X <- T, X >= Pivot]);
sort([]) -> [].
```

[X || X <- T, X < Pivot] means the list of X where X is taken from T and X is less than Pivot.

Adder

Higher order functions can be written as follows:

```
> Adder = fun(N) -> fun(X) -> X + N end end.
#Fun
> G = Adder(10).
#Fun
> G(5).
15
```

²⁴ The Erlang shell is an infinite read-eval-print loop.

²⁵ Similar to a LISP cons cell.

B.2 Primitives for concurrency

Spawn

```
Pid = spawn(fun() -> loop(0) end).
```

Send and receive

```
Pid ! Message,
.....
receive
    Message1 ->
        Actions1;
    Message2 ->
        Actions2;
    ...
    after Time ->
        TimeOutActions
end
```

B.3 Concurrent Erlang examples

“Area” server

```
-module(area).
-export([loop/1]).
```

```
loop(Tot) ->
    receive
        {Pid, {square, X}} ->
            Pid ! X*X,
            loop(Tot + X*X);
        {Pid, {rectangle,[X,Y]}} ->
            Pid ! X*Y,
            loop(Tot + X*Y);
        {Pid, areas} ->
            Pid ! Tot,
            loop(Tot)
    end.
```

“Area” client

```
Pid = spawn(fun() -> area:loop(0) end),
Pid ! {self(), {square, 10}},
receive
    Area ->
        ...
end
```

Global server

We can register a Pid so that we can refer to the process by a name:

```
...
Pid = spawn(Fun),
register(bank, Pid),
...
bank ! ...
```

B.4 Distributed Erlang

We can spawn a process on a remote node as follows:

```
...
Pid = spawn(Fun@Node)
...
alive(Node)
...
not_alive(Node)
```

B.5 Fault tolerant Erlang

catch

```
> X = 1/0.
** exited: {badarith, divide_by_zero} **
> X = (catch 1/0).
{'EXIT',{badarith, divide_by_zero}}
> b().
X = {'EXIT',{badarith, divide_by_zero}}
```

Catch and throw

```
case catch f(X) ->
    {exception1, Why} ->
        Actions;
    NormalReturn ->
        Actions;
end,
f(X) ->
...
Normal_return_value;
f(X) ->
...
throw({exception1, ...}).
```

Links and trapping exits

```
process_flag(trap_exits, true),
P = spawn_link(Node, Mod, Func, Args),
receive
    {'EXIT', P, Why} ->
        Actions;
    ...
end
```

B.6 Hot code replacement

Here's the inner loop of a server:

```
loop(Data, F) ->
    receive
        {request, Pid, Q} ->
            {Reply, Data1} = F(Q, Data),
            Pid ! Reply,
            loop(Data1, F);
        {change_code, F1} ->
            loop(Data, F1)
    end
```

To do a code replacement operation do something like:

```
Server ! {change_code, fun(I, J) ->
            do_something(...)
        end}
```

B.7 Generic client-server

The module `cs` is a simple generic client-server:

```
-module(cs).
-export([start/3, rpc/2]).
```

```
start(Name, Data, Fun) ->
    register(Name,
        spawn(fun() ->
            loop(Data, Fun)
        end)).
```

```
rpc(Name, Q) ->
    Tag = make_ref(),
    Name ! {request, self(), Tag, Q},
    receive
        {Tag, Reply} -> Reply
    end.

loop(Data, F) ->
    receive
        {request, Pid, Tag, Q} ->
            {Reply, Data1} = F(Q, Data),
            Pid ! {Tag, Reply},
            loop(Data1, F)
    end.

Parameterizing the server

We can parameterize the server like this:

-module(test).
-export([start/0, add/2, lookup/1]).

start() -> cs:start(keydb, [], fun handler/2).

add(Key, Val) -> cs:rpc(keydb, {add, Key, Val}).
lookup(Key) -> cs:rpc(keydb, {lookup, Key}).

handler({add, Key, Val}, Data) ->
    {ok, add(Key, Val, Data)};
handler({lookup, Key}, Data) ->
    {find(Key, Data), Data}.

add(Key, Val, [{Key, _}|T]) -> [{Key, Val}|T];
add(Key, Val, [H|T]) -> [H|add(Key, Val, T)];
add(Key, Val, []) -> [{Key, Val}].

find(Key, [{Key, Val}|_]) -> {found, Val};
find(Key, [H|T]) -> find(Key, T);
find(Key, []) -> error.
```

Here's a test run:

```
> test:start().
true
> test:add(xx, 1).
ok
> test:add(yy, 2).
ok
> test:lookup(xx).
{found,1}
> test:lookup(zz).
error
```

The client code (in `test.erl`) is *purely sequential*. Everything to do with concurrency (`spawn, send, receive`) is contained within `cs.erl`.

`cs.erl` is a simple *behavior* that hides the concurrency from the application program. In a similar manner we can encapsulate (and hide) error detection and recovery, code upgrades, etc. This is the basis of the OTP libraries.

Acknowledgments

I have mentioned quite a few people in this paper, but it would be a mistake to say that these were the only people involved in the Erlang story. A large number of people were involved and their individual and collective efforts have made Erlang what it is today. Each of these people has their own story to tell but unfortunately the space here does not allow me to tell these stories: all I can do is briefly mention them here. Thanks to all of you and to all the others whose names I have missed.

Mangement – Bjarne Däcker, Catrin Granbom, Torbjörn Johnson, Kenneth Lundin, Janine O’Keeffe, Mats Persson, Jane Walerud, Kerstin Ödling.

Implementation – Joe Armstrong, Per Bergqvist, Richard Carlsson, Bogumil (Bogdan) Hausman, Per Hedeland, Björn Gustavsson, Tobias Lindahl, Mikael Pettersson, Tony Rogvall, Kostis Sagonas Robert Tjärnström, Robert Virding, Klacke (Klacke) Wikström, Mike Williams.

Tools – Marcus Arendt, Thomas Arts, Johan Bevemyr, Martin Björklund, Hans Bolinder, Kent Boortz, Göran Båge, Micael Carlberg, Francesco Cesaroni, Magnus Fröberg, Joacim Grebenö, Luke Gorrie, Richard Green, Dan Gudmundsson, John Hughes, Bertil Karlsson, Thomas Lindgren, Simon Marlow, Håkan Mattsson, Hans Nilsson, Raimo Niskanen, Patrik Nyblom, Mickaël Rémond, Sebastian Strollo, Lars Thorsén, Torbjörn Törnvist, Karl-William Welin, Ulf Wiger, Phil Wadler, Patrik Winroth, Lennart Öhman.

Users – Ingela Anderton, Knut Bakke, Per Bergqvist, Johan Bevemyr, Ulf Svarte Bagge, Johan Blom, Maurice Castro, Tuula Carlsson, Mats Cronqvist, Anna Fedoriw, Lars-Åke Fredlund, Scott Lystig Fritchie, Dilian Gurov, Sean Hinde, Håkan Karlsson, Roland Karlsson, Håkan Larsson, Peter Lundell, Matthias Läng, Sven-Olof Nyström, Richard A. O’Keeffe, Erik Reitsma, Mickaël Rémond, Åke Rosberg, Daniel Schutte, Christopher Williams.

References

- [1] J.L. Armstrong. *Telephony Programming in Prolog*. Report T/SU 86 036 Ericsson Internal report. 3 March 1986.
- [2] J.L. Armstrong and M.C. Williams. *Using Prolog for rapid prototyping of telecommunication systems*. SETSS ’89 Bournemouth 3-6 July 1989
- [3] J.L. Armstrong, S.R. Virding and M.C. Williams. *Erlang User’s Guide and Reference Manual*. Version 3.2 Ellemtel Utvecklings AB, 1991.
- [4] J.L. Armstrong, S.R. Virding and M.C. Williams. *Use of Prolog for developing a new programming language*. Published in *The Practical Application of Prolog*. 1-3 April 1992. Institute of Electrical Engineers, London.
- [5] J.L. Armstrong, B.O. Däcker, S.R. Virding and M.C. Williams. *Implementing a functional language for highly parallel real-time applications*. Software Engineering for Telecommunication Switching Systems, March 30 - April 1, 1992 Florence.
- [6] Joe Armstrong. *Getting Erlang to talk to the outside world*. Proceedings of the 2002 ACM SIGPLAN workshop on Erlang.
- [7] Joe Armstrong. *Making reliable distributed systems in the presence of errors*. Ph.D. Thesis, Royal Institute of Technology, Stockholm 2003.
- [8] Staffan Blau and Jan Rooth. *AXD 301 – A new generation ATM switching system* Ericsson Review No. 1, 1998
- [9] D.L. Bowen, L. Byrd, and W.F. Clocksin, 1983. *A portable Prolog compiler*. Proceedings of the Logic Programming Workshop, Albufeira, Portugal, 74-83.
- [10] B. Däcker, N. Eishiewy, P. Hedeland, C-W. Welin and M.C. Williams. *Experiments with programming languages and techniques for telecommunications applications*. SETSS ’86. Eindhoven 14-18 April, 1986.
- [11] Bjarne Däcker. *Datalogilaboratoriet De första 10 åren*. Ellemtel Utvecklings AB, 1994.
- [12] Bjarne Däcker. *Concurrent Functional Programming for Telecomunications. A case study of technology introduction*: October 2000. Licentiate thesis. ISSN 1403-5286. Royal Institute of Technology. Stockholm, Sweden.
- [13] L. Peter, Deutsch, *A Lisp machine with very compact programs*. Proceedings of 3rd IJCAI, Stanford, Ca., Aug. 1973.
- [14] Ian Foster and Stephen Taylor. *Strand – New Concepts in Parallel Programming*. Prentice hall, 1990.
- [15] A.S. Haridi. *Logic Programs Based on a Natural Deduction System*. Ph.D. Thesis Royal Institute of Technology, Stockholm, 1981.
- [16] Bogumil Hausman. *Turbo Erlang: Approaching the speed of C*. In *Implementations of Logic Programming Systems*, Kluwer Academic Publishers, 1994.
- [17] S. Holmström. *A Functional Language for Parallel Programming*. Report No. 83.03-R. Programming Methodology Lab., Chalmers University of Technology, 1983.
- [18] Tobias Lindahl and Konstantinos Sagonas. *Detecting software defects in telecom applications through lightweight static analysis: A War Story*. APLAS 2004.
- [19] David Maier and David Scott Warren: *Computing with Logic: Logic Programming with Prolog*. Benjamin/Cummings 1988.
- [20] Simon Marlow Philip Wadler. *A practical subtyping system for Erlang*. ICFP 1997.
- [21] Håkan Mattsson, Hans Nilsson and Claes Wikström: *Mnesia – A distributed robust DBMS for telecommunications applications*. PADL 1999.
- [22] Eliot Miranda. *BrouHaHa – A portable Smalltalk interpreter*. SIGPLAN Notices 22 (12), December 1987 (OOPSLA ’87).
- [23] Paul Morrison. *Flow-Based Programming: A New Approach to Application Development*. Van Nostrand Reinhold, 1994.
- [24] Hans Nilsson, Torbjörn Törnvist and Claes Wikström: *Amnesia – A distributed real-time primary memory DBMS with a deductive query language*. ICLP 1995.
- [25] Hans Nilsson and Claes Wikström: *Mnesia – An industrial DBMS with transactions, distribution and a logical query language*. CODAS 1996.
- [26] Tommy Ringqvist. *BR Policy concerning the use of Erlang*. ERA/BR/TV-98:007. March 12, 1998. Ericsson Internal paper.
- [27] Tony Rogvall and Claes Wikström. *Protocol programming in Erlang using binaries*. Fifth International Erlang/OTP User Conference. 1999.
- [28] T. Sjöland. *Logic Programming with LPL0 – An Introductory Guide*. CSALAB The Royal Institute of Technology, Stockholm, Sweden. October 1983.
- [29] D.H.D. Warren. *An Abstract Prolog Instruction Set*. Technical Report 309, Artificial Intelligence Center, SRI International, 333 Ravenswood Ave, Menlo Park CA 94025, 1983.
- [30] Claes Wikström. *Distributed programming in Erlang*. PASCO’94. First International Symposium on Parallel Symbolic Computation.
- [31] Ulf Wiger. *Four-fold increase in productivity and quality – industrial strength functional programming in telecom-class products*. Workshop on Formal Design of Safety Critical Embedded Systems. March 21-23, 2001, Munich.

The Rise and Fall of High Performance Fortran: An Historical Object Lesson

Ken Kennedy Charles Koelbel

Rice University, Houston, TX

{ken,chk}@rice.edu

Hans Zima

Institute of Scientific Computing, University of
Vienna, Austria, and
Jet Propulsion Laboratory, California Institute of
Technology, Pasadena, CA

zima@jpl.nasa.gov

Abstract

High Performance Fortran (HPF) is a high-level data-parallel programming system based on Fortran. The effort to standardize HPF began in 1991, at the Supercomputing Conference in Albuquerque, where a group of industry leaders asked Ken Kennedy to lead an effort to produce a common programming language for the emerging class of distributed-memory parallel computers. The proposed language would focus on data-parallel operations in a single thread of control, a strategy which was pioneered by some earlier commercial and research systems, including Thinking Machines' CM Fortran, Fortran D, and Vienna Fortran.

The standardization group, called the High Performance Fortran Forum (HPFF), took a little over a year to produce a language definition that was published in January 1993 as a Rice technical report [50] and, later that same year, as an article in Scientific Programming [49].

The HPF project had created a great deal of excitement while it was underway and the release was initially well received in the community. However, over a period of several years, enthusiasm for the language waned in the United States, although it has continued to be used in Japan.

This paper traces the origins of HPF through the programming languages on which it was based, leading up to the standardization effort. It reviews the motivation underlying technical decisions that led to the set of features incorporated into the original language and its two follow-ons: HPF 2 (extensions defined by a new series of HPFF meetings) and HPF/JA (the dialect that was used by Japanese manufacturers and runs on the Earth Simulator).

Permission to make digital/hard copy of part of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date of appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Permission may be requested from the Publications Dept., ACM, Inc., 2 Penn Plaza, New York, NY 11201-0701, USA, fax:+1(212) 869-0481, permissions@acm.org

©2007 ACM 978-1-59593-766-7/2007/06-ART7 \$5.00

DOI 10.1145/1238844.1238851

<http://doi.acm.org/10.1145/1238844.1238851>

A unique feature of this paper is its discussion and analysis of the technical and sociological mistakes made by both the language designers and the user community; mistakes that led to the premature abandonment of the very promising approach employed in HPF. It concludes with some lessons for the future and an exploration of the influence of ideas from HPF on new languages emerging from the High Productivity Computing Systems program sponsored by DARPA.

Categories and Subject Descriptors K.2 History of Computing [*Software*]

General Terms Languages and Compilers, Parallel Computing

Keywords High Performance Fortran (HPF)

1. Background

Parallelism—doing multiple tasks at the same time—is fundamental in computer design. Even very early computer systems employed parallelism, overlapping input-output with computing and fetching the next instruction while still executing the current one. Some computers, like the CDC 6600, used multiple instruction execution units so that several long instructions could be in process at one time. Others used *pipelining* to overlap multiple instructions in the same execution unit, permitting them to produce one result every cycle even though any single operation would take multiple cycles. The idea of pipelining led to the first *vector computers*, exemplified by the Cray-1 [31], in which a single instruction could be used to apply the same operation to arrays of input elements with each input pair occupying a single stage of the operation pipeline. Vector machines dominated the supercomputer market from the late 1970s through the early 1980s.

By the mid-1980s, it was becoming clear that *parallel computing*, the use of multiple processors to speed up a single application, would eventually replace, or at least augment, vector computing as the way to construct leading-

edge supercomputing systems. However, it was not clear what the right high-level programming model for such machines would be. In this paper we trace the history of High Performance Fortran (HPF), a representative of one of the competing models, the *data-parallel* programming model. Although HPF was important because of its open and public standardization process, it was only one of many efforts to develop parallel programming languages and models that were active in the same general time period. In writing this paper, we are *not* attempting comprehensively to treat all this work, which would be far too broad a topic; instead we include only enough material to illustrate relationships between the ideas underlying HPF and early trends in parallel programming.

We begin with a narrative discussion of parallel computer architectures and how they influenced the design of programming models. This leads to the motivating ideas behind HPF. From there, we cover the standardization process, the features of the language, and experience with early implementations. The paper concludes with a discussion of the reasons for HPF's ultimate failure and the lessons to be learned from the language and its history.

1.1 Parallel Computer Systems

Several different types of parallel computer designs were developed during the formative years of parallel processing. Although we present these architectures in a sequential narrative, the reader should bear in mind that a variety of computers of all the classes were always available or in development at any given time. Thus, these designs should be viewed as both contemporaries and competitors.

Data-Parallel Computers The *data-parallel computation model* is characterized by the property that sequences of operations or statements can be performed in parallel on each element of a collection of data. Some of the earliest parallel machines developed in the 1960s, such as the Solomon [87] and Illiac IV [9] architectures, implemented this model in hardware. A single control unit executed a sequence of instructions, broadcasting each instruction to an array of simple processing elements arranged in a regular grid. The processing elements operated in lockstep, applying the same instruction to their local data and registers. Flynn [37] classifies these machines as *Single-Instruction Multiple-Data (SIMD)* architectures. Once you have a grid of processors, each with a separate local memory, data values residing on one processor and needed by another have to be copied across the interconnection network, a process called *communication*. Interprocessor communication of this sort causes long delays, or *latency*, for cross-processor data access.

The vector computers emerging in the 1970s, such as the Cray-1, introduced an architecture paradigm supporting a simple form of data parallelism in hardware. As with the original SIMD architectures, vector computers execute a sin-

gle thread of control; a key difference is the increased flexibility provided to the programmer by abandoning the need to arrange data according to a hardware-defined processor layout. Furthermore, a vector processor does not experience the problems of communication latency exhibited by more general SIMD processors because it has a single shared memory.

In the 1980s, advances in VLSI design led to another generation of SIMD architectures characterized by thousands of 1-bit processing elements, with hardware support for arbitrary communication patterns. Individual arithmetic operations on these machines were very slow because they were performed in “bit serial” fashion, that is, one bit a time. Thus any performance improvement came entirely from the high degrees of parallelism. Important representatives of this class of machines include the Connection Machines CM-2 and CM-200 [52] as well as the MasPar MP-1 [27].

The programming models for SIMD machines emphasized vector and matrix operations on large arrays. This was attractive for certain algorithms and domains (e.g., linear algebra solvers, operations on regular meshes) but confining in other contexts (e.g. complex Monte Carlo simulations). In particular, this model was seen as intuitive because it had a single thread of control, making it similar to the sequential programming model. As we will see, this model strongly influenced the design of data-parallel languages.

The principal strength of data-parallel computing—fully synchronous operation—was also its greatest weakness. The only way to perform conditional computations was to selectively turn off computing by some processors in the machine, which made data-parallel computers ill suited to problems that lacked regularity, such as sparse matrix processing and calculations on irregular meshes. An additional drawback was the framework for communicating data between processors. In most cases, the processors were connected only to nearest neighbors in a two- or three-dimensional grid.¹ Because all operations were synchronous, it could take quite a long time to move data to distant processors on a grid, making these machines slow for any calculation requiring long-distance communication. These drawbacks led to the ascendance of asynchronous parallel computers with more flexible interconnection structures.

Shared-Memory Asynchronous Parallel Computers A key step in the emergence of today’s machines was the development of architectures consisting of a number of full-fledged processors, each capable of independently executing a stream of instructions. At first, the parallel programming community believed that the best design for such *Multiple-Instruction Multiple-Data (MIMD)* multiprocessor systems (in the Flynn classification) should employ some form of hardware shared memory, because it would make it easy to implement a shared-memory programming model, which

¹ Important exceptions include STARAN [10] with its “flip” network, the CM-1, CM-2 and CM-200 [52] with their hypercube connections, and the MasPar MP-1 [27] with its Global Router.

was generally viewed as the most natural from the programmer's perspective. In this model, the independent processors could each access the entire memory on the machine. A large number of shared-memory multiprocessors, which are today referred to as *symmetric multiprocessors (SMPs)*, appeared as commercial products. Examples of such systems include the Alliant FX series, CDC Cyber 205, Convex C series, Cray XMP and YMP, Digital VAX 8800, Encore Multimax, ETA-10, FLEX/32, IBM 3090, Kendall Square KSR1 and KSR2, Myrias SPS-1 and SPS-2, and the Sequent Balance series [5, 30, 97, 56, 59, 62, 71, 74, 48, 32]. It should be noted that several of the machines in this list (e.g., the Convex, Cray, ETA, and IBM systems) were hybrids in the sense that each processor could perform vector operations. Many of today's multi-core architectures can be considered descendants of SMPs.

In the late 1980s, the conventional view was that shared-memory multiprocessors would be easy to program while providing a big performance advantage over vector computers. Indeed, many thought that shared memory multiprocessing would yield computing power that was limited only by how much the user was willing to pay. However, there were two problems that needed to be resolved before this vision could be realized.

The first problem was *scalability*: how such systems could be scaled to include hundreds or even thousands of processors. Because most of the early SMPs used a bus—a single multi-bit data channel that could be multiplexed on different time steps to provide communication between processors and memory—the total bandwidth to memory was limited by the aggregate number of bits that could be moved over the bus in a given time interval. For memory-intensive high-performance computing applications, the bus typically became saturated when the number of processors exceeded 16. Later systems would address this issue through the use of crossbar switches, but there was always a substantive cost, either in the expense of the interconnect or in loss of performance of the memory system.

The second problem was presented by the programming model itself. Most vendors introduced parallel constructs, such as the *parallel loop*, that, when used incorrectly, could introduce a particularly nasty form of bug called a *data race*. A data race occurs whenever two parallel tasks, such as the iterations of a parallel loop, access the same memory location, with at least one of the tasks writing to that location. In that case, different answers can result from different parallel schedules. These bugs were difficult to locate and eliminate because they were not repeatable; as a result debuggers would need to try every schedule if they were to establish the absence of a race. Vector computers did not suffer from data races because the input languages all had a single thread of control: it was up to the compiler to determine when the operations in a program could be correctly expressed as vector instructions.

Distributed-Memory Computers The scalability problems of shared memory led to a major change in direction in parallel computation. A new paradigm, called *distributed-memory parallelism* (or more elegantly, *multicomputing*), emerged from academic (the Caltech Cosmic Cube [85] and Suprenum [41]) and commercial research projects (Transputer networks [92]). In distributed-memory computers, each processor was packaged with its own memory and the processors were interconnected with networks, such as two-dimensional and three-dimensional meshes or hypercubes, that were more scalable than the bus architectures employed on shared-memory machines.

Distributed memory had two major advantages. First, the use of scalable networks made large systems much more cost-effective (at the expense of introducing additional latency for data access). Second, systems with much larger aggregate memories could be assembled. At the time, almost every microprocessor used 32-bit (or smaller) addresses. Thus a shared-memory system could only address 2^{32} different data elements. In a distributed-memory system, on the other hand, each processor could address that much memory. Therefore, distributed memory permitted the solution of problems requiring much larger memory sizes.

Unsurprisingly, the advantages of distributed memory came at a cost in programming complexity. In order for one processor to access a data element in another processor, the processor in whose local memory the data element was stored would need to *send* it to the processor requesting it; in turn, that processor would need to *receive* the data before using it. Sends and receives had to be carefully synchronized to ensure that the right data was communicated, as it would be relatively easy to match a receive with the wrong send. This approach, which was eventually standardized as *Message Passing Interface (MPI)* [70, 88, 42], requires that the programmer take complete responsibility for managing and synchronizing communication. Thus, the move to distributed memory sacrificed the convenience of shared memory while retaining the complexity of the multiple threads of control introduced by SMPs.

As a minor simplification, the standard programming model for such systems came to be the *Single-Program Multiple Data (SPMD)* model [58, 34], in which each processor executed the same program on different portions of the data space, typically that portion of the data space owned by the executing processor. This model, which is an obvious generalization of the SIMD data-parallel model, required the implementation to explicitly synchronize the processors before communication because they might be working on different parts of the program at any given time. However, it allows the effective use of control structures on a per-processor basis to handle processor-local data in between communication steps and permits the programmer to focus more narrowly on communications at the boundaries of each processor's data space. Although this is a major improvement, it does

not completely eliminate the burden of managing communication by hand.

1.2 Software Support for Parallel Programming

In general, there were three software strategies for supporting parallel computing: (1) automatic parallelization of sequential languages, (2) explicitly parallel programming models and languages, and (3) data-parallel languages, which represented a mixture of strategies from the first two. In the paragraphs that follow we review some of the most important ideas in these three strategies with a particular emphasis on how they were used to support data parallelism, the most common approach to achieving scalability in scientific applications.

Automatic Parallelization The original idea for programming shared-memory machines was to adapt the work on automatic vectorization which had proven quite successful for producing vectorized programs from sequential Fortran² specifications [3, 99]. This work used the theory of *dependence* [65, 63, 4], which permitted reasoning about whether two distinct array accesses could reference the same memory location, to determine whether a particular loop could be converted to a sequence of vector assignments. This conversion involved distributing the loop around each of the statements and then rewriting each loop as a series of vector-register assignments. The key notion was that any statement that depended on itself, either directly or indirectly, could not be rewritten in this manner.

Vectorization was an extremely successful technology because it focused on inner loops, which were easier for programmers to understand. Even though the so-called “dusty-deck Fortran” programs did not always vectorize well, the user could usually rewrite those codes into “vectorizable loop” form, which produced good behavior across a wide variety of machines. It seemed very reasonable to assume that similar success could be achieved for shared-memory parallelization.

Unfortunately, SMP parallelism exhibited additional complexities. While vector execution was essentially synchronous, providing a synchronization on each operation, multiprocessor parallelism required explicit synchronization operations (e.g. barriers or event posting and waiting), in addition to the costs of task startup and the overhead of data sharing across multiple caches. To compensate for these costs, the compiler either needed to find very large loops that could be subdivided into large chunks, or it needed to parallelize outer loops. Dependence analysis, which worked well for vectorization, now needed to be applied over much larger loops, which often contained subprogram invocations. This led researchers to focus on interprocedural analysis as a strategy

² Although the official standards changed the capitalization from “FORTRAN” to “Fortran” beginning with Fortran 90, we use the latter form for all versions of the language.

for determining whether such loops could be run in parallel [93, 16, 29].

By using increasingly complex analyses and transformations, research compilers have been able to parallelize a number of interesting applications, and commercial compilers are routinely able to parallelize programs automatically for small numbers of processors. However, for symmetric multiprocessors of sufficient scale, it is generally agreed that some form of user input is required to do a good job of parallelization.

The problems of automatic parallelization are compounded when dealing with distributed memory and message-passing systems. In addition to the issues of discovery of parallelism and granularity control, the compiler must determine how to lay out data to minimize the cost of communication. A number of research projects have attacked the problem of automatic data layout [67, 60, 6, 23, 44], but there has been little commercial application of these strategies, aside from always using a standard data layout for all arrays (usually block or block-cyclic).

Explicitly Parallel Programming: PCF and OpenMP Given the problems of automatic parallelization, a faction of the community looked for simple ways to specify parallelism explicitly in an application. For Fortran, the first examples appeared on various commercial shared-memory multiprocessors in the form of parallel loops, parallel cases, and parallel tasks. The problem with these efforts was lack of consistency across computing platforms. To overcome this, a group convened by David J. Kuck of the University of Illinois and consisting of researchers and developers from academia, industry, and government laboratories began to develop a standard set of extensions to Fortran 77 that would permit the specification of both loop and task parallelism. This effort came to be known as the *Parallel Computing Forum (PCF)*. Ken Kennedy attended all of the meetings of this group and was deeply involved in drafting the final document that defined PCF Fortran [66], a single-threaded language that permitted SPMD parallelism within certain constructs, such as parallel loops, parallel case statements, and “parallel regions.” A “parallel region” created an SPMD execution environment in which a number of explicitly parallel loops could be embedded. PCF Fortran also included mechanisms for explicit synchronization and rules for storage allocation within parallel constructs.

The basic constructs in PCF Fortran were later standardized by ANSI Committee X3H5 (Parallel Extensions for Programming Languages) and eventually found their way into the informal standard for OpenMP [73, 33].

The main limitation of the PCF and OpenMP extensions is that they target platforms—shared-memory multiprocessors with uniform memory access times—that have been eclipsed at the high end by distributed-memory systems. Although it is possible to generate code for message-passing systems from OpenMP or PCF Fortran, the user has no way

of managing communication or data placement at the source level, which can result in serious performance problems.

Data-parallel Languages One way to address the shortcomings of the PCF/OpenMP model is explicit message passing. In this approach, a standard language (such as Fortran or C/C++) is extended with a message-passing library (such as MPI), providing the programmer with full control over the partitioning of data domains, their distribution across processors, and the required communication. However, it was soon understood that this programming paradigm can result in complex and error-prone programs due to the way in which algorithms and communication are inextricably interwoven.

This led to the question of whether the advantages of shared memory, and even a single thread of control, could be *simulated* on a distributed-memory system. A second important question was how parallelism could be made to scale to hundreds or thousands of processors. It was clear that addressing the second question would require exploiting the data-parallel programming model: by subdividing the data domain in some manner and assigning the subdomains to different processors, the degree of parallelism in a program execution is limited only by the number of processors, assuming the algorithm provides enough parallelism. With more available processors, a larger problem can be solved.

These issues led researchers to explore the new class of *data-parallel languages*, which were strongly influenced by the SIMD programming paradigm and its closeness to the dominating sequential programming model. In data-parallel languages, the large data structures in an application would be laid out across the memories of a distributed-memory parallel machine. The subcomponents of these distributed data structures could then be operated on in parallel on all the processors. Key properties of these languages include a global name space and a single thread of control through statements at the source level, with individual parallel statements being executed on all processors in a (loosely) synchronous manner.³ Communication is not explicitly programmed, but automatically generated by the compiler/runtime system, based on a declarative specification of the data layout.

Any discussion of data-parallel languages should include a discussion of Fortran 90, because it was the first version of Fortran to include array assignment statements. A Fortran 90 array assignment was defined to behave as if the arrays used on the right-hand side were all copied into unbounded-length vector registers and operated upon before any stores occurred on the left. If one considers the elements of an infinite-length vector register as a distributed memory, then in a very real sense, a Fortran 90 array assignment is a data-parallel operation. In addition, if you had a large SIMD machine, such as the Thinking Machines CM-2, multidimensional array assignments could be also be executed in paral-

lel. It is for this reason that Fortran 90 was so influential on the data-parallel languages to follow. However, with the advent of more complex distributed-memory systems, achieving good Fortran 90 performance became more challenging, as the data distributions needed to take both load balance and communication costs into account, along with the limitations of available storage on each processor.

A number of new data-parallel languages were developed for distributed-memory parallel machines in the late 1980s and early 1990s, including *Fortran D* [38, 53], *Vienna Fortran* [102, 21], *CM Fortran* [91], *C** [45], *Data-Parallel C*, *pC++* [14] and *ZPL* [89, 19]. Several other academic as well as commercial projects also contributed to the understanding necessary for the development of HPF and the required compilation technology [7, 47, 79, 68, 69, 77, 78, 80, 81, 95, 57, 51, 75].

To be sure, the data-parallel language approach was not universally embraced by either the compiler research or the application community. Each of the three strategies for software support described in this section had their passionate adherents. In retrospect, none of them has prevailed in practice: explicit message-passing using MPI remains the dominant programming system for scalable applications to this day. The main reason is the complexity of compiler-based solutions to the parallelization, data layout, and communication optimization problems inherent, to varying degrees, in each of the three strategies. In the rest of this paper, we narrow our focus to the data-parallel approach as embodied in the HPF extensions to Fortran. In the discussion that follows, we explore the reasons for the failure of HPF in particular. However, many of the impediments to the success of HPF are impediments to the other approaches as well.

2. HPF and Its Precursors

In the late 1980s and early 1990s, Fortran was still the dominant language for technical computing, which in turn was the largest market for scalable machines. Therefore, it was natural to assume that a data-parallel version of Fortran would be well received by the user community, because it could leverage the enormous base of software written in that language. Two research languages, Fortran D and Vienna Fortran, and one commercial product, CM Fortran, were among the most influential data-parallel languages based on Fortran and deeply influenced the development of HPF. In this section we review the salient issues in each of these languages. The section then concludes with a description of the activities leading to the HPF standardization effort, along with an overview of the effort itself.

Fortran D In 1987, the research group led by Ken Kennedy at Rice began collaborating with Geoffrey Fox on how to support high-level programming for distributed-memory computers. Fox had observed that the key problem in writing an application for a distributed-memory system was to choose the right data distribution, because once that was

³ A computation is said to be *loosely synchronous* if it consists of alternating phases of computation and interprocessor communication.

done, the actual parallelism was determined by the need to minimize communications. Thus, computations would be done in parallel on the processors that owned the data involved in that computation. This led to the idea that a language with shared memory and a single thread of control could be compiled into efficient code for a distributed-memory system if the programmer would provide information on how to distribute data across the processors. Fox and the Rice group produced a specification for a new language called Fortran D that provided a two-level distribution specification similar to that later adopted into HPF. The basic idea was that groups of arrays would be aligned with an abstract object called a *template*, then all of these arrays would be mapped to processors by a single distribution statement that mapped the template to those processors. Fortran D supported *block*, *cyclic*, and *block-cyclic* (sometimes called *cyclic(k)*) distribution of templates onto processors in multiple dimensions. For each distinct distribution, a different template needed to be specified. However, this mechanism could be used to ensure that arrays of different sizes, such as different meshes in multigrid calculations, would be mapped to the right processors.

Of course, a data-parallel language like Fortran D would not be useful unless it could be compiled to code with reasonable performance on each parallel target platform. The challenge was to determine how to decompose the computations and map them to the processors in a way that would minimize the cost of communication. In addition, the compiler would need to generate communication when it was necessary and optimize that communication so that data was moved between pairs of processors in large blocks rather than sequences of single words. This was critical because, at the time, most of the cost of any interprocessor communication operation was the time to deliver the first byte (latencies were very large but bandwidth was quite reasonable). Fortran D was targeted to communication libraries that supported two-sided protocols: to get data from one processor to another, the owning processor had to send it, while the processor needing the data had to receive it. This made communication generation complicated because the compiler had to determine where both sends and receives were to be executed on each processor.

To address the issue of computation partitioning, the Rice project defined a strategy called “owner-computes”, which locates computations on processors near the data where it is stored. In particular, the first Rice compiler prototypes⁴ used “left-hand-side owner-computes”, which compiles each statement so that all computations are performed on the processors owning the computation outputs.

⁴ Strictly speaking the Fortran D compilers, and most research implementations of data-parallel languages were source-to-source translators that generated SPMD implementation, such as Fortran plus message-passing calls as the “object code.” In fact, one (minor) motivation for initiating the MPI standardization effort was to provide a machine-independent target for data-parallel compilers.

The first compilation paper, by Callahan and Kennedy, describing the Rice strategy appeared in the 1988 LCPC conference at Cornell and was included in a collection from that conference in the Journal of Supercomputing [18]. (Earlier in the same year, Zima, Bast and Gerndt published their paper on the SUPERB parallelization tool [100], which also used a distribution-based approach.) Although Callahan and Kennedy described the owner-computes strategy in rudimentary form, the optimizations of communication and computation were performed locally, using transformations adapted from traditional code optimization. When this approach proved ineffective, the Rice group switched to a strategy that compiled whole loop nests, one at a time. This work was described in a series of papers that also covered the prototype implementation of this new approach [54, 53, 94]. Although they presented results on fairly small programs, these papers demonstrated substantive performance improvements that established the viability of distribution-based compilation. At the heart of the compilation process is a conversion from the global array index space provided in Fortran D to a local index space on each processor. To perform this conversion the compiler would discover, for each loop, which iterations required no communication, which required that data be sent to another processor, and which required that data be received before any computation could be performed. A summary of this approach appears in Chapter 14 of the book by Allen and Kennedy [4].

One important issue had to be dealt with in order to generate correct code for Fortran D over an entire program: how to determine, at each point in the program, what distribution was associated with each data array. Since distributions were not themselves data objects, they could not be explicitly passed to subprograms; instead they were implicitly associated with data arrays passed as parameters. To generate code for a subprogram, the compiler would have to follow one of three approaches: (1) perform some kind of whole program analysis of distribution propagation, (2) rely on declarations by the programmer at each subroutine interface, or (3) dynamically determine the distributions at runtime by inspecting a descriptor for each distributed array. The Fortran D group decided that dynamic determination would be too slow and relying on the programmer would be impractical, particularly when libraries might be written in the absence of the calling program. As a result, the Fortran D compiler performed an interprocedural analysis of the propagation of data distributions, making it a whole-program compiler from the outset. This simplified the issues at procedure boundaries, but complicated the compiler structure in a way that would later prove unpalatable to the HPF standardization group.

Vienna Fortran In 1985, a group at Bonn University in Germany, led by Hans Zima, started the development of a new compilation system, called SUPERB, for a data-parallel language in the context of the German Suprenum

supercomputing project [41]. SUPERB [100] took as input a Fortran 77 program and a specification of a generalized block data distribution, producing an equivalent explicitly parallel message-passing program for the Suprenum distributed-memory architecture using the owner-computes strategy. This approach, which originally was interpreted as performing a computation on the processor owning the left-hand side of an assignment statement (or, more generally, the target of the computation), was later generalized to the selection of *any* processor that would maximize the locality of the computation. This turned out to be one of the most important ideas of HPF. Gerndt's Ph.D. dissertation [40], completed in 1989, is the first work describing in full detail the program transformations required for such a translation; an overview of the compilation technology is presented by Zima and Chapman in [101].

SUPERB was not a language design project: it focused on compiler transformations, using an ad-hoc notation for data distribution. However, after Zima's group relocated to University of Vienna, they began working, in collaboration with Piyush Mehrotra of NASA ICASE, on a full specification of a high-level data distribution language in the context of Fortran.

This new language, called Vienna Fortran, provided the programmer with a facility to define arrays of virtual processors, and introduced distributions as mappings from multidimensional array index spaces to (sub)sets of processor spaces. Special emphasis was placed on support for irregular and adaptive programs: in addition to the regular *block* and *block-cyclic* distribution classes the language provided *general block* and *indirect* distributions. *General block* distributions, inherited from SUPERB, partition an array dimension into contiguous portions of arbitrary lengths that may be computed at run time. Such distributions, when used in conjunction with reordering, can efficiently represent partitioned irregular meshes. *Indirect* distributions present another mechanism for the support of irregular problems by allowing the specification of arbitrary mappings between array index sets and processors. Both the general block and indirect distributions were later incorporated in the HPF 2.0 specification, described in Section 5. Implementing either of these distributions is difficult because the actual mappings are not known until run time. Therefore, the compiler must generate a preprocessing step, sometimes called an *inspector* [26, 96], that determines at run time a communication schedule and balances the loads across the different processors.

A key component of the Vienna Fortran language specification was a proposal for user-defined distributions and alignments. Although this feature was only partially implemented, it motivated research in the area of distributed sparse matrix representations [96] and provided important ideas for a recent implementation of such concepts in the Chapel high productivity language [17]. The Vienna Fortran Compilation

System extended the functionality of the SUPERB compiler to cover most of the language, while placing strong emphasis on the optimization of irregular algorithms. An overview of the compilation technology used in this system is presented in Benkner and Zima [12].

CM Fortran CM Fortran was the premier commercial implementation of a data-parallel language. The language was developed by a team led by Guy Steele at Thinking Machines Corporation (makers of the Connection Machines) and a group of implementers led by David Loveman and Robert Morgan of COMPASS, Inc., a small software company. The goal of CM Fortran was to support development of technical applications for the CM-2, a SIMD architecture introduced in 1987. The original programming model for the CM-2 and its forerunner, the CM-1, had been *Lisp because the initial market had focused on applications in artificial intelligence. However, a Fortran compiler was released in 1991 [91], with the goal of attracting science and engineering users.

CM Fortran adopted the array assignment and array arithmetic extensions that had been incorporated into Fortran 90, but it also included a feature that was deleted from the Fortran 90 standard at the last minute: the FORALL statement, which permitted a particularly simple loop-like specification for array assignments. Arrays that were used in these statements were classified as CM arrays and mapped to virtual processor (VP) sets, one array element per processor. VPs were in turn mapped to the actual processors of the CM-2 in regular patterns. Optional ALIGN and LAYOUT directives could modify this mapping. Unlike Fortran D, CM Fortran used compiler directives, entered as comments, to specify data layouts for the data arrays in the CM-2 SIMD processor array. Each array assignment was compiled into a sequence of SIMD instructions to compute execution masks, move data, and invoke the actual computation. This was a simple and effective strategy that was extended in the Thinking Machines "slicewise" compiler strategy to reduce redundant data movement and masks.

Programmability in CM Fortran was enhanced by the availability of a rich library of global computation and communication primitives known as CMSSL, which was developed under the leadership of Lennart Johnsson. In addition to powerful computational primitives, it included a number of global operations, such as sum reduction, and scatter/gather operations, that are used to convert data in irregular memory patterns to compact arrays and vice versa. Many of these routines were later incorporated into the HPF Library specification.

When Thinking Machines introduced a MIMD architecture, the CM-5, in 1993, it retained the CM Fortran language for the new architecture, showing the popularity and portability of the model.

The HPF Standardization Process In November of 1991 at Supercomputing '91 in Albuquerque, New Mexico, Kennedy

and Fox were approached about the possibility of standardizing the syntax of data-parallel versions of Fortran. The driving forces behind this effort were the commercial vendors, particularly Thinking Machines, who were producing distributed-memory scalable parallel machines. Digital Equipment Corporation (DEC) was also interested in producing a cross-platform for Fortran that included many of the special features included in DEC Fortran.

In response to the initial discussions, Kennedy and Fox met with a number of academic and industrial representatives in a birds-of-a-feather session. The participants agreed to explore a more formal process through a group that came to be known as the High Performance Fortran Forum (HPFF). Kennedy agreed to serve as the HPFF chair and Charles Koelbel assumed the role of Executive Director. With support from the Center for Parallel Computation Research (CRPC) at Rice University, a meeting was hurriedly organized in Houston, Texas, in January 1992. Interest in the process was evident from the overflow attendance (nearly 100 people) and active discussions during presentations. The meeting concluded with a business session in which more than 20 companies committed to a process of drafting the new standard.

Because the interest level was high and the need pressing, it was agreed that the process should try to produce a result in approximately one year. Although we did not fully realize it at the time, this “management” agreement would affect the features of the language. The timeline forced us to use the informal rule that HPF would adopt only features that had been demonstrated in at least one language and compiler (including research projects’ compilers). This of course limited some of the features considered, particularly in the realm of advanced data distributions. We have to admit, however, that the “at least one compiler” rule was, as Shakespeare might have said, oft honored in the breach. In particular, the committee felt that no fully satisfactory mechanism for subroutine interfaces had been demonstrated, and therefore fashioned a number of complementary features.

The active HPFF participants (30-40 people) met for two days every six weeks or so, most often in a hotel in Dallas, Texas, chosen because of convenience to airline connections.⁵ There was a remarkable collection of talented individuals among the regular attendees. Besides Kennedy and Koelbel, those participants who were editors of the standard document included Marina Chen, Bob Knighten, David Loveman, Rob Schreiber, Marc Snir, Guy Steele, Joel Williamson, and Mary Zosel. Table 1 contains a more complete list of HFFF attendees, ordered by their affiliations at the time. We include the affiliations for two reasons: each organization (that had been represented at two of the past

⁵In part by design, the same hotel was also used for meetings of the Message Passing Interface Forum (MPIF) which took place a bit later. One participant in both forums from England later joked that it was the only hotel in the world where he could order “the usual” and get the right drink.

three meetings) had one vote, and the affiliations illustrate the breadth of the group. There are representatives from hardware and software vendors, university computer science researchers, and government and industrial application users. HPFF was a consensus-building process, not a single-organization project.

This group dealt with numerous difficult technical and political issues. Many of these arose because of the tension between a need for high language functionality and the desire for implementation simplicity. It was clear from the outset that HPF would need to be flexible and powerful enough to implement a broad range of applications with good performance. However, real applications differ significantly in the way they deal with data and in their patterns of computation, so different data distributions would be needed to accommodate them. Each data distribution built in to the language required a lot of effort from the compiler developers. So how many data distributions would be enough? As a case in point, consider the the *block-cyclic* distribution in which groups of k rows or columns of an array are assigned in round-robin fashion to the processors in a processor array. From the experience with the research compilers, the implementation of this distribution was unquestionably going to be challenging. Some members of the HPF Forum argued against it. In the end, however, it was included because it was needed to balance the computational load across processors on triangular calculations such as those done in dense linear algebra (e.g., LU Decomposition). In general, the need for *load balancing*, which attempts to assign equal amounts of computation to each processor for maximum speedup, motivated the need for a rich set of distribution patterns.

The problems of power versus complexity were compounded by limited experience with compilation of data-parallel languages. The research compilers were mostly academic prototypes that had been applied to few applications of any size. CM Fortran, on the other hand, was relatively new and was not as feature-rich as either Fortran D or Vienna Fortran. Therefore many decisions had to be made without full understanding of their impact on the complexity of the compilers.

Another difficult decision was whether to base the language on Fortran 77 or Fortran 90, the new standard as of the beginning of the HPF process. The problem was that most of the companies making parallel machines had compilers that handled Fortran 77 only. They were reluctant to commit to implementing the full Fortran 90 standard as a first step toward HPF. On the other hand, the companies that had already based their compilers on Fortran 90, such as Thinking Machines, wanted to take advantage of the language advances.

In the end, the users participating in the process tipped the balance toward Fortran 90. There were many reasons for this. Primary among them was that Fortran 90 included features that would make the HPF language definition cleaner, particularly at subprogram interfaces. In addition, Fortran 90

Table 1. Attendees (and institutions at the time) in HPFF process, 1992-1993

David Reese (Alliant)	Jerrold Wagener (Amoco)
Rex Page (Amoco)	John Levesque (APR)
Rony Sawdayi (APR)	Gene Wagenbreth (APR)
Jean-Laurent Philippe (Archipel)	Joel Williamson (Convex Computer)
David Presberg (Cornell Theory Center)	Tom MacDonald (Cray Research)
Andy Meltzer (Cray Research)	David Loveman (Digital)
Siamak Hassanzadeh (Fujitsu America)	Ken Muira (Fujitsu America)
Hidetoshi Iwashita (Fujitsu Laboratories)	Clemens-August Thole (GMD)
Maureen Hoffert (Hewlett Packard)	Tin-Fook Ngai (Hewlett Packard)
Richard Schooler (Hewlett Packard)	Alan Adamson (IBM)
Randy Scarborough (IBM)	Marc Snir (IBM)
Kate Stewart (IBM)	Piyush Mehrotra (ICASE)
Bob Knighten (Intel)	Lev Dyadkin (Lahey Computer)
Richard Fuhler (Lahey Computer)	Thomas Lahey (Lahey Computer)
Matt Snyder (Lahey Computer)	Mary Zosel (Lawrence Livermore)
Ralph Brickner (Los Alamos)	Margaret Simmons (Los Alamos)
J. Ramanujam (Louisiana State)	Richard Swift (MasPar Computer)
James Cownie (Meiko)	Barry Keane (nCUBE)
Venkata Konda (nCUBE)	P. Sadayappan (Ohio State)
Robert Babb II (OGI)	Vince Schuster (Portland Group)
Robert Schreiber (RIACS)	Ken Kennedy (Rice)
Charles Koelbel (Rice)	Peter Highnam (Schlumberger)
Don Heller (Shell)	Min-You Wu (SUNY Buffalo)
Prakash Narayan (Sun)	Douglas Walls (Sun)
Alok Choudhary (Syracuse)	Tom Haupt (Syracuse)
Edwin Paalvast (TNO-TU Delft)	Henk Sips (TNO-TU Delft)
Jim Bailey (Thinking Machines)	Richard Shapiro (Thinking Machines)
Guy Steele (Thinking Machines)	Richard Shapiro (United Technologies)
Uwe Geuder (Stuttgart)	Bernhard Woerner (Stuttgart)
Roland Zink (Stuttgart)	John Merlin (Southampton)
Barbara Chapman (Vienna)	Hans Zima (Vienna)
Marina Chen (Yale)	Aloke Majumdar (Yale)

included an array operation syntax that was consistent with the concept of global arrays. Such operations would make it easier to identify global operations that could be carried out in parallel. In retrospect, this decision may have been a mistake, because the implementation of Fortran 90 added too much complexity to the task of producing a compiler in a reasonable time frame. As we point out in Section 4, this may have contributed to the slow development of HPF compilers and, hence, to the limited acceptance of the language.

Another major technical issue was how to handle distribution information in subroutines. Developers of technical applications in Fortran had come to rely on the ability to build on libraries of subroutines that were widely distributed by academic institutions and sold by commercial enterprises such as IMSL and NAG. If HPF was to be successful, it should be possible to develop libraries that worked correctly on array parameters with different distributions. Thus, it should be possible to determine within a subroutine what data distributions were associated with the arrays passed into that subroutine.

Fortran D had dealt with this issue by mandating an interprocedural compiler. The vendors in the HPF Forum were understandably unwilling to follow this route because most commercial compilers included no interprocedural analysis capabilities (leaving aside the question of whether library vendors would be willing to provide source in the first place). Thus the language design needed a way to declare distributions of parameters so that HPF subroutine libraries could be used with different distributions, without suffering enormous performance penalties. A key issue here was whether to redistribute arrays at subroutine boundaries or somehow “inherit” the distribution from the calling program. Redistribution adds data movement costs on entry and exit to a subroutine, but allows the routine to employ a distribution that is optimized to the underlying algorithm. Inheriting the caller’s distribution avoids redistribution costs, but may have performance penalties due to a distribution that is inappropriate for the algorithm or the cost of dynamically interpreting the distribution that is passed in. This discussion involved the most complex technical issues in the standardization process and

the parts of the standard dealing with these issues are the most obscure in the entire document.

This example illustrates one of the problems with the HPF standardization process and the tight schedule that was adopted. Although the members agreed at the outset only to include features that had been tried in some prototype research or commercial implementation, the group sometimes ignored this guideline, substituting intuition for experience.

The result of this work was a new language that was finalized in early 1993 [50, 49] and presented at the Supercomputing Conference in the fall of the same year. The specifics of the language are discussed in the next section.

An additional set of HPFF meetings were held during 1994 with the goals of (1) addressing needed corrections, clarifications, and interpretations of the existing standard and (2) considering new features required in extending HPF to additional functions. The organization (including place and frequency) of the meetings was as in the first series. Ken Kennedy again served as Chair, while Mary Zosel took on the role of Executive Director. The attendees were mostly the same as in the 1992-1993 meetings, with the notable additions of Ian Foster (Argonne National Laboratory) and Joel Saltz (then at the University of Maryland), who each led subgroups considering future issues. The corrections were incorporated in the High Performance Fortran 1.1 standard, a slight revision of the 1993 document presented at the Supercomputing '94 conference in Washington, DC in November 1994. The HPF 1.1 document itself refers to some of the issues needing clarification as "dark corners" of the language (those issues do not affect the examples in the next section). The new features and clarifications discussed but not included in HPF 1.1 were collected in the HPF Journal of Development and served as a starting point for the HPF 2 standardization effort, discussed in Section 5.

3. The HPF Language

The goals established for HPF were fairly straightforward:

- To provide convenient programming support for scalable parallel computer systems, with a particular emphasis on data parallelism
- To present an accessible, machine-independent programming model with three main qualities: (1) The application developer should be able to view memory as a single shared address space, even on distributed-memory machines; in other words, arrays should be globally accessible but distributed across the memories of the processors participating in a computation. (2) Programs written in the language should appear to have a single thread of control, so that the program could be executed correctly on a single processor; thus, all parallelism should derive from the parallel application of operations to distributed data structures. (3) Communication should be implicitly generated, so that the programmer need not be

concerned with the details of specifying and managing inter-processor message passing.

- To produce code with performance comparable to the best hand-coded MPI for the same application.

To achieve these goals the HPF 1.0 Standard defined a language with a number of novel characteristics.

First, the language was based on Fortran 90, with the extensions defined as a set of "directives" in the form of Fortran 90 comments. These directives could be interpreted by HPF compilers as advice on how to produce a parallel program. On a scalar machine, an HPF program could be executed without change by simply ignoring the directives, assuming the machine had sufficient memory. This device, which was later copied by the OpenMP standards group, permitted the HPF parallelism extensions to be separated cleanly from the underlying Fortran 90 program: the program still needed to embody a data-parallel algorithm, but the same program should work on both sequential and parallel systems. Compilers for the sequential systems would not be required to recognize any HPF directives. It should be noted that the ability to run essentially the same program on both a sequential and parallel machine is a huge advantage for the application programmer in debugging an algorithm.

The principal additions to the language were a set of *distribution directives*, that specified how arrays were to be laid out across the memories of the machine. Sets of arrays could be aligned with one another and then distributed across the processors using built-in distributions, such as *block*, *cyclic*, and *block-cyclic*. These directives could be used to assign the individual rows or columns to processors in large blocks, or smaller blocks in round-robin fashion. We illustrate this by showing the application of directives to a simple relaxation loop:

```
REAL A(1000,1000), B(1000,1000)
DO J = 2, N
  DO I = 2, N
    A(I,J)=(A(I,J+1)+2*A(I,J)+A(I,J-1))*0.25 &
    &           + (B(I+1,J)+2*B(I,J)+B(I-1,J))*0.25
  ENDDO
ENDDO
```

The *DISTRIBUTE* directive specifies how to partition a data array onto the memories of a real parallel machine. In this case, it is most natural to distribute the first dimension, since iterations over it can be performed in parallel. For example, the programmer can distribute data in contiguous chunks across the available processors by inserting the directive

```
!HPF$ DISTRIBUTE A(BLOCK,*)
```

after the declaration of A. HPF also provides other standard distribution patterns, including *CYCLIC* in which elements are assigned to processors in round-robin fashion, or *CYCLIC(K)* by which blocks of K elements are assigned round-robin to processors. Generally speaking, *BLOCK* is

the preferred distribution for computations with nearest-neighbor elementwise communication, while the CYCLIC variants allow finer load balancing of some computations. Also, in many computations (including the example above), different data arrays should use the same or related data layouts. The ALIGN directive specifies an elementwise matching between arrays in these cases. For example, to give array B the same distribution as A, the programmer would use the directive

```
!HPF$ ALIGN B(I,J) WITH A(I,J)
```

Integer linear functions of the subscripts are also allowed in ALIGN and are useful for matching arrays of different shapes.

Using these directives, the HPF version of the example code is:

```
REAL A(1000,1000), B(1000,1000)
!HPF$ DISTRIBUTE A(BLOCK,*)
!HPF$ ALIGN B(I,J) WITH A(I,J)
DO J = 2, N
  DO I = 2, N
    A(I,J)=(A(I,J+1)+2*A(I,J)+A(I,J-1))*0.25 &
    &           + (B(I+1,J)+2*B(I,J)+B(I-1,J))*0.25
  ENDDO
ENDDO
```

Once the data layouts have been defined, implicit parallelism is provided by the *owner-computes* rule, which specifies that calculations on distributed arrays should be assigned in such a way that each calculation is carried out on the processors that own the array elements involved in that calculation. Communication would be implicitly generated when a calculation involved elements from two different processors.

As the Fortran D project and Vienna Fortran projects showed, data distribution of subroutine arguments was a particularly complex area. To summarize the mechanism that HPF eventually adopted, formal subroutine arguments (i.e. the variables as declared in the subroutine) could have associated ALIGN and DISTRIBUTE directives. If those directives fully specified a data distribution, then the actual arguments (i.e. the objects passed by the subroutine caller) would be redistributed to this new layout when the call was made, and redistributed back to the original distribution on return. Of course, if the caller and callee distributions matched, it was expected that the compiler or runtime system would forego the copying needed in the redistribution. HPF also defined a system of “inherited” distributions by which the distribution of the formal arguments would be identical to the actual arguments. This declaration required an explicit subroutine interface, such as a Fortran 90 INTERFACE block. In this case, no copying would be necessary, but code generation for the subroutine would be much more complex to handle all possible incoming distributions. This complexity was so great, in fact, that to our knowledge no compiler fully implemented it.

In addition to the distribution directives, HPF has special directives that can be used to assist in the identification of parallelism. Because HPF is based on Fortran 90, it also has array operations to express elementwise parallelism directly. These operations are particularly appropriate when applied to a distributed dimension, in which case the compiler can (relatively) easily manage the synchronization and data movement together. Using array notation in this example produces the following:

```
REAL A(1000,1000), B(1000,1000)
!HPF$ DISTRIBUTE A(BLOCK,*)
!HPF$ ALIGN B(I,J) WITH A(I,J)
DO J = 2, N
  A(2:N,J) = &
  &   (A(2:N,J+1)+2*A(2:N,J)+A(I,J-1))*0.25 &
  &   + (B(3:N+1,J)+2*B(2:N,J)+B(1:N-1,J))*0.25
ENDDO
```

In addition to these features, HPF included the ability to specify that the iterations of a loop should be executed in parallel. Specifically, the INDEPENDENT directive says that the loop that follows is safe to execute in parallel. This can be illustrated with the code from the example above.

```
REAL A(1000,1000), B(1000,1000)
!HPF$ DISTRIBUTE A(BLOCK,*)
!HPF$ ALIGN B(I,J) WITH A(I,J)
DO J = 2, N
  !HPF$ INDEPENDENT
  DO I = 2, N
    A(I,J)=(A(I,J+1)+2*A(I,J)+A(I,J-1))*0.25 &
    &           + (B(I+1,J)+2*B(I,J)+B(I-1,J))*0.25
  ENDDO
ENDDO
```

Use of the directive ensures that a parallel loop will be generated by any HPF compiler to which the program is presented. Many compilers can detect this fact for themselves when analyzing programs with subscript expressions that are linear in the loop variables (as in the above example), based on dependence analysis along with the distribution information. However, the INDEPENDENT directive is essential for loops that are theoretically unanalyzable—for example, loops iterating over the edges of an unstructured mesh, which contain subscripted subscripts. Often the programmer will have application-specific knowledge that allows such loops to be executed in parallel.

Although the INDEPENDENT directive violates the goal of presenting a single thread of control, the issue was sidestepped in the standard by defining the directive as an assertion that the loop had no inter-iteration dependencies that would lead to data races; if this assertion was incorrect, the program was declared to be not *standard-conforming*. Thus, a standard-conforming HPF program would always produce the same answers on a scalar machine as a parallel one. Un-

fortunately, this feature made it impossible to determine at compile time whether a program was standard-conforming.

HPF also provided the FORALL statement, taken from CM Fortran and early drafts of Fortran 90, as an alternative means of expressing array assignment. The nested DO loop in our relaxation example could be written as

```
FORALL (J = 2:N, I=2:N) &
&   A(I,J)=(A(I,J+1)+2*A(I,J)+A(I,J-1))*0.25 &
&           + (B(I+1,J)+2*B(I,J)+B(I-1,J))*0.25
```

Semantically, the FORALL was identical to an array assignment; it computed the values on the right-hand side for all index values before storing the results into any left-hand side location. (There was also a multi-statement FORALL that applied this semantic rule to all assignments in the body in turn.) The explicit indexing allowed FORALL to conveniently express a wider range of array shapes and computations than the standard array assignment, as in the following example.

```
! Assignment to a diagonal, computed its index
FORALL (I=1:N) A(I,I) = I*I
```

High Performance Fortran was one of the first languages to include the specification for an associated library, the *HPF Library*, as a part of the defined language. Special global operations, such as sum reduction, gather and scatter, and partial prefix operations were provided by the HPF Library, which incorporated many parallel operations on global arrays that proved to be useful in other data-parallel languages, such as CM Fortran. This library added enormous power to the language. Specification of an associated library is now standard practice in C, C++, and Java.

Finally, HPF included a number of features that were designed to improve compatibility and facilitate interoperation with other programming languages and models. In particular, the EXTRINSIC interface made it possible to invoke subprograms that were written in other languages such as scalar Fortran and C. Of particular importance was the ability to call subroutines written in MPI in a way that made it possible to recode HPF subprograms for more efficiency.

4. Experience with the Language

The initial response to HPF could be characterized as cautious enthusiasm. A large part of the user community, those who had not already recoded using explicit message-passing, was hopeful that the language would permit a high-level programming interface for parallel machines that would make parallel programs portable and efficient without the need for extensive coding in MPI or its equivalent. The vendors, on the other hand, were hoping that HPF would expand the market for scalable parallel computing enough to increase profitability. Several vendors initiated independent compiler efforts, including Digital [46], IBM [43], and Thinking Machines. A number of other hardware vendors offered OEM versions of compilers produced by independent software vendors such as the Portland Group, Inc.

(PGI) [15] and Applied Parallel Research [8]. At its peak, there were 17 vendors offering HPF products and over 35 major applications written in HPF, at least one of which was over 100,000 lines of code.

Nevertheless, as experience with the language increased, so did frustration on the part of the users. It became clear that it was not as easy as had been hoped to achieve high performance and portability in the language and many application developers gave up and switched to MPI. By the late 1990s usage of HPF in the United States had slowed to a trickle, although interest in Japan remained high, as we discuss below.

Given that HPF embodied a set of reasonable ideas on how to extend an existing language to incorporate data parallelism, why did it not achieve more success? In our view there were four main reasons: (1) inadequate compiler technology, combined with a lack of patience in the HPC community; (2) insufficient support for important features that would make the language suitable for a broad range of problems; (3) the inconsistency of implementations, which made it hard for a user to achieve portable performance; and (4) the complex relationship between program and performance, which made performance problems difficult to identify and eliminate. In the paragraphs that follow, we explore each of these issues in more detail.

Immature Compiler Technology When HPF was first released, Kennedy gave numerous addresses to technical audiences cautioning them to have limited expectations for the first HPF compiler releases. There were many reasons why caution was appropriate.

First, HPF was defined on top of Fortran 90, the first major upgrade to the Fortran standard since 1977. Furthermore, the Fortran 90 extensions were not simple: implementing them would require an enormous effort for the compiler writers. Among the new features added to Fortran 90 were (1) mechanisms for whole and partial array handling that required the use of descriptor-based implementations and “scalarization” of array assignments, (2) modules and interface blocks (which HPF depended on for specification of the distributions of arrays passed to subprograms), (3) recursion, and (4) dynamic storage allocation and pointer-based data structures. Since none of these features were present in Fortran 77, building a Fortran 90 compiler required a substantive reimplementation to introduce stack frames for recursion, heap storage, and array descriptors. Moving from Fortran 77 to Fortran 90 involved an effort comparable to building a compiler from scratch (except for the low-level code generation). At the time of the first HPF specification in 1993, most companies had not yet released their first Fortran 90 compilers. Thus, to produce an implementation of HPF, those companies would first need to implement almost all of Fortran 90, putting a huge obstacle in the way of getting to HPF.

Second, the entire collection of features in HPF, including the HPF library, was quite extensive and required new compilation strategies that, at the time of the release of HPF 1.0, had only been implemented in research compilers and the CM Fortran compiler. Proper compilation of HPF requires extensive global analysis of distributions, partitioning of computation, generation of communication, and optimizations such as overlapping communication and computation. Implementing all of these well would require compilers to mature over a number of years, even if implementing Fortran 90 were not a precondition.

Finally, efficient implementation of HPF programs required that the compiler pay special attention to locality on individual processors. Since most of the processors used in distributed-memory systems were uniprocessors with complex cache hierarchies, generating efficient code required that advanced transformation strategies, such as tiling for cache reuse, be employed. At the time of the release of the initial HPF specification, these techniques were beginning to be understood [39, 98, 64], but most commercial compilers had not yet incorporated them [84, 28].

In spite of this, the HPC community was impatient for a high-level programming model, so there was heavy pressure on the compiler vendors to release some implementation of HPF. As a result the first compilers were premature, and the performance improvements they provided were disappointing in all but the simplest cases.

Meanwhile, the developers of parallel applications had deadlines to meet: they could not afford to wait for HPF to mature. Instead they turned to Fortran with explicit MPI calls, a programming model that was complex but was, at least, ready to use. The migration to MPI significantly reduced the demand for HPF, leading compiler vendors to reduce, or even abandon, the development effort. The end result was that HPF never achieved a sufficient level of acceptance within the leading-edge parallel computing users to make it a success. In a very real sense, HPF missed the first wave of application developers and was basically dead (or at least considered a failure) before there was a second wave.

Missing Features To achieve high performance on a variety of applications and algorithms, a parallel programming model must support a variety of different kinds of data distributions. This is particularly critical for sparse data structures or adaptive algorithms. The original specification of HPF included only three main distributions: BLOCK, CYCLIC, and CYCLIC(K). These distributions are effective for dense array computations and, in the case of CYCLIC(K), even linear algebra. However, many important algorithmic strategies were difficult, or even impossible, to express within HPF without a big sacrifice of performance. This deficiency unnecessarily narrowed the space of applications that could be effectively expressed in HPF. As a result, the developers of applications requiring distributions that were not supported went directly

to MPI. Although this problem was partially corrected in HPF 2.0 (discussed later), the damage was already done. In the final section of this paper, we suggest a strategy for addressing this problem in future data-parallel languages.

A second issue with HPF was its limited support for task parallelism. The parallel loop feature helped a little, but users wanted more powerful strategies for task parallelism. Once again, this was corrected in HPF 2.0, but it was too late. What should have happened was a merger between the features in HPF and OpenMP, discussed below.

Barriers to Achieving Portable Performance One of the key goals of HPF was to make it possible for an end user to have one version of a parallel program that would then produce implementations on different architectures that would achieve a significant fraction of the performance possible on each architecture. This was not possible for two main reasons.

First, different vendors focused on different optimizations in their HPF implementations. This caused a single HPF application to achieve dramatically different performance on the machines from different vendors. In turn, this led users to recode the application for each new machine to take advantage of the strengths (and avoid the weaknesses) of each vendor's implementation, thwarting the original goal of absolute portability.

Second, the HPF Library could have been used to address some of the usability and performance problems described in previous sections. For example, the “gather” and “scatter” primitives could have been used to implement sparse array calculations. However, there was no open-source reference implementation for the library, so it was left to each compiler project to implement its own version. Because of the number and complexity of the library components, this was a significant implementation burden. The end result was that too little attention was paid to the library and the implementations were inconsistent and exhibited generally poor performance. Thus, users were once again forced to code differently for different target machines, using those library routines that provided the best performance.

Difficulty of Performance Tuning Every HPF compiler we are aware of translated the HPF source to Fortran plus MPI. In the process, many dramatic transformations were carried out, making the relationship between what the developer wrote and what the parallel machine executed somewhat murky. This made it difficult for the user to identify and correct performance problems.

To address the identification problem, the implementation group at Rice collaborated with Dan Reed's group at the University of Illinois to map the Pablo Performance Analysis Systems diagnostics, which were based on a message-passing architecture, back to HPF source [2]. This effort was extremely effective and was implemented in at least one commercial compiler, but it did little to address the tuning problem. That is, the user could well understand what was

causing his or her performance problem, but have no idea how to change the HPF source to overcome the issue. Of course, he or she could use the EXTRINSIC interface to drop into MPI, but that voided the advantages of using HPF in the first place.

The identification problem in the context of performance tuning was also addressed in a cooperation between the Vienna Fortran group and Maria Calzarossa's research group at the University of Pavia, Italy. This project developed a graphical interface in which the explicitly parallel MPI-based target code, with performance information delivered by the MEDEA tool, could be linked back to the associated source statements. For example, a programmer using this tool and familiar with the code generated for an independent loop in the framework of the *inspector/executor* paradigm [83] was able to analyze whether the source of a performance problem was the time required for the distribution of work, the (automatic) generation of the communication schedule by the inspector, or the actual communication generated for the loop.

5. The HPF 2 Standardization Effort

In an attempt to correct some of the deficiencies in HPF 1.0 and 1.1, the HPF Forum undertook a second standardization effort from 1995 to 1996. This effort led to the HPF 2.0 standard which incorporated a number of new features:

1. The REDUCTION clause for INDEPENDENT loops, substantially expanding the cases where INDEPENDENT could be used. (HPF 1.0 had the NEW clause for INDEPENDENT to allow "local" variables to each loop iteration, but no straightforward way to allow simple accumulations.)
2. The new HPF_LIBRARY procedures SORT_DOWN, SORT_UP, to perform sorting. (HPF 1.0 had already introduced the GRADE_UP and GRADE_DOWN functions, which produced permutation vectors rather than sorting elements directly.)
3. Extended data mapping capabilities, including mapping of objects to *processor subsets*; mapping of pointers and components of derived types; the GEN_BLOCK and INDIRECT distribution patterns and the RANGE and SHADOW modifiers to distributions. (HPF 1.0, as noted above, was limited to very regular distribution patterns on array variables.)
4. Extended parallel execution control, including the ON directive to specify the processor to execute a computation, the RESIDENT directive to mark communication-free computations, and the TASK_REGION directive providing coarse-grain parallel tasks. (HPF 1.0 left all computation mapping to the compiler and runtime system.)
5. A variety of additional intrinsic and HPF_LIBRARY procedures, mostly concerned with querying and managing

data distributions. (HPF 1.0 had some support, but more was found necessary.)

6. Support for asynchronous I/O with a new statement WAIT, and an additional I/O control parameter in the Fortran READ/WRITE statement. (HPF 1.0 had ignored I/O facilities.)

Except for the first two items above, these new features were "Approved Extensions" rather than the "Core Language". HPFF had intended that vendors would implement all of the core language features (e.g. the REDUCTION clause) immediately, and prioritize the extensions based on customer demand. Not surprisingly, this created confusion as the feature sets offered by different vendors diverged. To our knowledge, no commercial vendor or research project ever attempted an implementation of the full set of approved extensions.

6. The Impact and Influence of HPF

Although HPF has not been an unqualified success, its has been enormously influential in the development of high-level parallel languages. The current CiteSeer database lists 827 citations for the original 1993 technical report, which was later published in Scientific Programming [50], making it the 21st most cited document in the computer science field (as covered by CiteSeer). In addition, over 1500 publications in CiteSeer refer to the phrase "High Performance Fortran". Many of these papers present various approaches to implementing the language or improving upon it, indicating that it generated a great deal of intellectual activity in the academic community.

6.1 Impact on Fortran and its Variants

Fortran 95 While the meetings that led to HPF 1.1 were underway, the X3J3 committee of ANSI was also meeting to develop the Fortran 95 standard [1]. That group had long watched developments in HPF with an eye toward adopting successful features, with Jerry Wagener serving as an informal liaison between the groups. Ken Kennedy gave a presentation to X3J3 on HPF's parallel features, and ensured that no copyright issues would hamper incorporation of HPF features into the official Fortran standard. When the Fortran 95 standard [1] was officially adopted in 1996, it included the HPF FORALL and PURE features nearly verbatim. The new standard also included minor extensions to MAXLOC and MINLOC that had been adopted into the HPF Library from CM Fortran. Both HPFF and X3J3 considered this sharing a positive development.

HPF/JA In 1999, the Japan Association for High Performance Fortran, a consortium of Japanese companies including Fujitsu, Hitachi, and NEC, released HPF/JA [86], which included a number of features found in previous programming languages on parallel-vector machines from Hitachi and NEC. An important source contributing to HPF/JA was

the HPF+ language [11] developed and implemented in a European project led by the Vienna group, with NEC as one of the project partners. HPF+, resulting from an analysis of advanced industrial codes, provided a REUSE clause for independent loops that asserted reusability of the communication schedule computed during the first execution of the loop. In the same context, the HALO construct of HPF+ allowed the functional specification of nonlocal data accesses in processors and user control of the copying of such data to region boundaries.

These and other features allowed for better control over locality in HPF/JA programs. For example, the LOCAL directive could be used to specify that communication was not needed for data access in some situations where a compiler would find this fact difficult to discern. In addition, the REFLECT directive included in HPF/JA corresponds to HPF+'s HALO feature. HPF/JA was implemented on the Japanese Earth Simulator [86], discussed below.

OpenMP Finally, we comment on the relationship between HPF and OpenMP [73, 33]. OpenMP was proposed as an extension of Fortran, C, and C++, providing a set of directives that support a well-established portable shared memory programming interface for SMPs based on a fork/join model of parallel computation. It extends earlier work performed by the Parallel Computing Forum (PCF) as part of the X3H5 standardization committee [66] and the SGI directives for shared memory programming. OpenMP followed the HPF model of specifying all features in the form of directives, which could be ignored by uniprocessor compilers. Most of the OpenMP features were completely compatible with HPF: in fact, the parallel loop constructs were basically extensions of the HPF parallel loop construct.

OpenMP is an explicitly parallel programming model in which the user is able to generate threads to utilize the processors of a shared memory machine and is also able to control accesses to shared data in an efficient manner. Yet, the OpenMP model does not provide features for expressing the mapping of data to processors in a distributed-memory system, nor does it permit the specification of processor/thread affinity. Such a paradigm will work effectively as long as locality is of no concern.

This shortcoming was recognized early, and a number of attempts have been made to integrate OpenMP with language features that allow locality-aware programming [72, 13]. However, the current language standard does not support any of these proposals. In fact, the recent development of *High Productivity Languages* (see Section 6.2) may render such attempts obsolete, since all these languages integrate multithreading and locality awareness in a clean way.

As a final remark, it is interesting to note that the OpenMP design group, which started its work after the release of HPF 1.0, defined syntax for its directives that was incompatible with HPF syntax. (From the OpenMP side, it was probably equally puzzling that HPF had defined its own syntax

rather than adopt the existing PCF constructs.) The language leaders of both efforts met on several occasions to explore the possibility of unifying the languages, as this would have provided needed functionality to each, but a combined standardization project never got started.

HPF Usage After the initial release of High Performance Fortran, there were three meetings of the HPF Users Group: one in Santa Fe, NM (February 24-26, 1997), the second in Porto, Portugal (June 25-26, 1998) and most recently in Tokyo, Japan (October 18-20, 2000). At these meetings surveys were taken to determine usage and product development. The papers presented at the Tokyo meeting were collected into a special double issue of *Concurrency, Practice and Experience* (Volume 14, Number 8-9, August 2002). The Tokyo meeting was remarkable in demonstrating the continuing high interest in HPF within Japan. This was reemphasized later when the Japanese Earth Simulator [86] was installed and began running applications. The Earth Simulator featured a high-functionality HPF implementation, initially based on the Portland Group compiler, that supported the HPF/JA extensions. Two applications described in the CPE special issue were brought up on the Earth Simulator and achieved performance in the range of 10 Teraflops and above. The application Impact3D ran at nearly 15 Teraflops, or roughly 40 percent of the peak speed of the machine [82]; it was awarded a Gordon Bell Prize at SC2002 for this achievement. Neither of these applications included any MPI.

6.2 HPCS Languages

Although HPF missed the leading edge of parallel application developers, its ideas are finding their way into newer programming languages. Recently, DARPA has funded three commercial vendors to develop prototype hardware and software systems for the High Productivity Computing Systems (HPCS) projects. A principal goal of this effort is to ease the burden of programming leading-edge systems that are based on innovative new architectures. The three vendors have produced three new language proposals: Chapel (Cray) [17, 20, 36], Fortress (Sun) [90], and X10 (IBM) [22]. All of these include data parallelism in some form.

Chapel, Fortress, and X10 are new object-oriented languages supporting a wide range of features for programmability, parallelism, and safety. They all provide a global name space, explicit multithreading, and explicit mechanisms for dealing with locality. This includes features for distributing arrays across the computing nodes of a system, and establishing affinity between threads and the data they are operating upon. Finally, these languages support both data and task parallelism. Because the HPCS language designers were familiar with the HPF experience, they have constructed data-parallel features that address many of the shortcomings described in Section 4.

In contrast to the directive-oriented approach of HPF or the library-based specification of MPI, Chapel, X10, and Fortress are *new* memory-managed object-oriented languages supporting a wide range of safety features. They build upon the experiences with object-oriented languages and their implementation technology over the past decade, but try to eliminate well-known shortcomings, in particular with respect to performance. At the same time, they integrate key ideas from many parallel languages and systems, including HPF.

X10 is based on Java; however, Java's support for concurrency and arrays has been replaced with features more appropriate for high performance computing such as a partitioned global address space. In contrast, Chapel is a completely new language based on what are perceived as the most promising ideas in a variety of current object-oriented approaches.

Locality Management All three languages provide the user with access to virtual units of locality, called respectively *locales* in Chapel, *regions* in Fortress, or *places* in X10. Each execution of a program is bound to a set of such locality units, which are mapped by the operating system to physical entities, such as computational nodes. This provides the user with a mechanism to (1) distribute data collections across locality units, (2) align different collections of data, and (3) establish affinity between computational threads and the data they operate upon. This approach represents an obvious generalization of key elements in HPF.

Fortress and X10 provide extensive libraries of built-in distributions, with the ability to produce new user-specified data distributions by decomposing the index space or combining distributions in different dimensions. In Fortress, all arrays are distributed by default; if no distribution is specified, then an array is assigned the default distribution. Chapel, on the other hand, has no built-in distributions but provides an extensive framework supporting the specification of arbitrary user-defined distributions that is powerful enough to deal with sparse data representations [36]. X10 has a *Locality Rule* that disallows the direct read/write of a remote reference in an activity. Chapel and Fortress do not distinguish in the source code between local and remote references.

Multithreading HPF specifies parallel loops using the *independent* attribute, which asserts that the loop does not contain any loop-carried dependences, thus excluding data races. Chapel distinguishes between a sequential *for* loop and a parallel *forall* loop, which iterates over the elements of an index domain, without a restriction similar to HPF's *independent* attribute. Thus the user is responsible for avoiding dependences that lead to data races.

The Fortress for-loop is parallel by default, so if a loop iterates over a distributed dimension of an array the iterations will be grouped onto processors according to the distributions. A special “sequential” distribution can be used

to serialize a for-loop. The Fortress compiler and run-time system are free to rearrange the execution to improve performance so long as the meaning of the program under the distribution-based looping semantics is preserved. In particular, the subdivision into distributions essentially overpartitions the index space so that operations can be dynamically moved to free processors to optimize load balance.

X10 distinguishes two kinds of parallel loops: the *foreach* loop, which is restricted to a single locality unit, and the *ateach* loop that allows iteration over multiple locality units. As with the *Locality Rule* discussed above, X10 pursues a more conservative strategy than Chapel or Fortress, forcing the programmer to distinguish between these two cases.

6.3 Parallel Scripting Languages

Although the Fortran and C communities were willing to tolerate the difficulties of writing MPI code for scalable parallel machines, it seems unlikely that the large group of users of high-level scripting languages such as Matlab, R, and Python will be willing to do the same. Part of the reason for the popularity of these languages is their simplicity.

Nevertheless, there is substantive interest in being able to write parallel code in these languages. As a result, a number of research projects and commercial endeavors, including The MathWorks, have been exploring strategies for parallel programming, particularly in Matlab [76, 35, 55, 61]. Most of these projects replace the standard Matlab array representation with a global distributed array and provide replacements for all standard operators that perform distributed operations on these arrays. Although this is in the spirit of HPF, the overhead of producing operation and communication libraries by hand limits the number of different distributions that can be supported by such systems. Most of the current implementations are therefore restricted to the distributions that are supported by ScalAPACK, the parallel version of LAPACK, which is used to implement array operations in the Matlab product.

A recently initiated project at Rice, led by Kennedy, is seeking to build on the Rice HPF compiler technology, described in Section 7 below, to provide a much richer collection of array distributions. The basic idea behind this effort is to produce the distributed array operations in HPF and allow the HPF compiler to specialize these library routines to all the different supported distributions. Currently, this list includes sequential(*), block, block-cyclic, and block(k), in each dimension, plus a new, two-dimensional distribution called “multipartitioning” that is useful for achieving the best possible performance on the NAS Parallel Benchmarks. However, a number of new distributions are being contemplated for the near future.

The goal of these efforts is to provide the scripting language community, and especially Matlab users, with a simple way to get reasonably scalable parallelism with only minimal change to their programs. If they are successful, they will not only vindicate the vision behind HPF, but will

also dramatically increase the community of application developers for scalable parallel machines.

7. Lessons Learned

For the future, there are some important lessons to be learned from the HPF experience: in particular, what should be done differently in future languages with similar features. In this section, we discuss a few of these issues.

First, we have learned a great deal about compiler technology in the twelve years since HPF 1.0 was released. It is clear that, on the first compilers, high performance was difficult to achieve in HPF. In fact, it became common practice to recode application benchmarks to exploit strengths of the HPF compiler for a particular target machine. In other words, application developers would rewrite the application for each new language platform. A case in point was the NAS parallel benchmark suite. A set of HPF programs that were coded by engineers at the Portland group was included as the HPF version of the NAS benchmarks. These versions were designed to get the highest possible performance on the Portland Group compiler, avoiding pitfalls that would compromise the efficiency of the generated code. Unfortunately, this compromised the HPF goal of making it possible to code the application in a machine-independent form and compile it without change to different platforms. In other words, the practice of coding to the language processor undermined HPF's ease of use, making it time consuming to port from one platform to another.

Over the past decade a research project at Rice University led by John Mellor-Crummey has been focused on the goal of achieving high performance on HPF programs that are minimally changed from the underlying Fortran 90. In the case of the NAS Parallel Benchmarks, this would mean that the sequential Fortran 90 program (embodiment a potential parallel algorithm) would be changed only by the addition of appropriate distribution directives: the compiler should do the rest. The ultimate goal would be to produce code that was competitive with the hand-coded MPI versions of the programs that were developed for the NAS suite.

The HPF compiler produced by the Rice project, dHPF, was not able to achieve this goal because the MPI versions used a distribution, called *multipartitioning*, that was not supported in either HPF 1.0 or 2.0. When the Rice researchers added multipartitioning to the distributions supported by dHPF, they were able to achieve performance that was within a few percentage points of the hand-coded MPI versions for both the NAS SP and BT applications [25]. This result is confirmed by experience with HPF on the Earth Simulator and a subsequent experiment in which the dHPF compiler translated Impact3D to run on the Pittsburgh Supercomputer Center's Lemieux system (based on the Alpha processor plus a Quadrics switch) and achieved over 17 percent efficiency on processor counts from 128 to 1024 (the

latter translated to 352 Gigaflops, likely a U.S. land speed record for HPF programs) [24].

The experience with multipartitioning illustrates another important aspect of making data-parallel languages more broadly applicable: there needs to be some way to expand the number of distributions available to the end user. Unfortunately, multipartitioning is but one of many possible distributions that one might want to add to an HPF compiler; the HPF 2.0 generalized block and indirect distributions are two others. There may be many other distributions that could be used to achieve high performance with a particular application on a particular parallel system. The base system cannot possibly include every useful distribution. This suggests the need for some mechanism for adding user-defined distributions. If we were able to define an interface for distributions and then define compiler optimizations in terms of that interface, we would have succeeded in separating data structure from data distribution. The end result would be a marked increase in the flexibility of the language. Research in this area is currently performed in the context of the Chapel programming language developed in the Cascade project. Chapel [17] provides no built-in distributions but offers a distribution class interface allowing the explicit specification of mappings, the definition of sequential and parallel iterators, and, if necessary, the control of the representation of distributions and local data structures.

As we indicated earlier, the differences in implementation strengths of the base language, combined with the paucity of good implementations of the HPF Library, were another reason for the limited success of the language. There were two dimensions to this problem. First, some companies never implemented the library at all. Others chose to spend their resources on specific routines that were needed by their customers and provide less than optimally efficient versions of the rest. The result was that the end user could not rely on the HPF Library if efficiency were a concern, which reduced the usability of the language overall.

One of the reasons for the success of MPI was the existence of a pretty good portable reference implementation called MPICH. A similar portable reference implementation for both the HPF compiler and the HPF Library would have given a significant boost to the language. The Portland Group compiler came pretty close to a reference implementation for the languages, but there was no corresponding standardized implementation for the library. Several leaders of the HPF implementer community discussed the possibility of securing Federal support for a project to provide a reference implementation of the HPF library but the funding was never found. This was particularly frustrating because there existed a very good implementation of most of the needed functionality in CMSSL (Connection Machine Scientific Subroutine Library), part of the Thinking Machines CMFortran run-time system. When Thinking Machines went out of business and was sold to Sun Microsystems, it might

have been possible to spin out the needed library functionality had there been government support, but this second opportunity was also missed.

Performance tuning tools for HPF presented another set of problems. As indicated in our earlier discussion, a collaboration between compiler and tool developers could succeed in mapping performance information back to HPF source: the Pablo system did this with the Rice dHPF compiler and the Portland group compiler [2]. The difficulty was that there were only limited ways for a user to exercise fine-grained control over the code generated once the source of performance bottlenecks was identified, other than using the EXTRINSIC interface to drop into MPI. The HPF/JA extensions ameliorated this a bit by providing more control over locality. However, it is clear that additional features are needed in the language design to override the compiler actions where that is necessary. Otherwise, the user is relegated to solving a complicated inverse problem in which he or she makes small changes to the distribution and loop structure in hopes of tricking the compiler into doing what is needed.

As a final note, we should comment on the lack of patience of the user community. It is true that High Performance Fortran was rushed into production prematurely, but had people been willing to keep the pressure up for improvement rather than simply bolting to MPI, we could have persevered in providing at least one higher-level programming model for parallel computing. As the work at Rice and in Japan has shown, HPF can deliver high performance with the right compiler technology, particularly when features are provided that give the end user more control over performance. Unfortunately, the community in the United States was unwilling to wait. The United States federal government could have helped stimulate more patience by funding one or more of the grand-challenge application projects to use new tools like HPF, rather than just getting the application to run as fast as possible on a parallel platform. Because of this lack of patience and support, the second wave of parallel computing customers is unable to benefit from the experiences of the first. This is a lost opportunity for the HPC community.

8. Conclusion

High Performance Fortran failed to achieve the success we all hoped for it. The reasons for this were several: immature compiler technology leading to poor performance, lack of flexible distributions, inconsistent implementations, missing tools, and lack of patience by the community. Nevertheless, HPF incorporated a number of important ideas that will be a part of the next generation of high performance computing languages. These include a single-threaded execution model with a global address space, interoperability with other language models, and an extensive library of primitive operations for parallel computing. In addition, a decade of research and development has overcome many of the implementation impediments. Perhaps the time is right for the

HPC community to once again embrace new data-parallel programming models similar to the one supported by HPF.

References

- [1] Jeanne C. Adams, Walter S. Brainard, Jeanne T. Martin, Brian T. Smith, and Jerrold L. Wagener. *Fortran 95 Handbook. Complete ISO/ANSI Reference*. Scientific and Engineering Computation Series. MIT Press, Cambridge, Massachusetts, 1997.
- [2] Vikram S. Adve, John Mellor-Crummey, Mark Anderson, Ken Kennedy, Jhy-Chun Wang, and Daniel A. Reed. An Integrated Compilation and Performance Analysis Environment for Data Parallel Programs. In *Proceedings of Supercomputing '95*, November 1995.
- [3] J. R. Allen and K. Kennedy. Automatic Translation of Fortran Programs to Vector Form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.
- [4] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, San Francisco, California, 2002.
- [5] Alliant Computer Systems. <http://en.wikipedia.org/wiki/Alliant-Computer-Systems>.
- [6] Jennifer M. Anderson and Monica S. Lam. Global Optimizations for Parallelism and Locality on Scalable Parallel Machines. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 112–125, New York, NY, USA, 1993. ACM Press.
- [7] F. Andre, J.-L. Pazat, and H. Thomas. PANDORE: A System to Manage Data Distribution. In *International Conference on Supercomputing*, pages 380–388, Amsterdam, The Netherlands, June 1990.
- [8] Applied Parallel Research, Sacramento, California. *Forge High Performance Fortran xhpf User's Guide, Version 2.1*, 1995.
- [9] G. H. Barnes, R. M. Brown, M. Kato, D. Kuck, D. Slotnick, and R. Stokes. The ILLIAC IV computer. *IEEE Transactions on Computers*, C-17:746–757, 1968.
- [10] Kenneth E. Batcher. The Multi-Dimensional Access Memory in STARAN. *IEEE Transactions on Computers*, C-26(2):174–177, February 1977.
- [11] S. Benkner, G. Lonsdale, and H.P. Zima. The HPF+ Project: Supporting HPF for Advanced Industrial Applications. In *Proceedings EuroPar'99 Parallel Processing*, volume 1685 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.
- [12] S. Benkner and H. Zima. Compiling High Performance Fortran for Distributed-Memory Architectures. *Parallel Computing*, 1999.
- [13] J. Birsak, P. Craig, R. Crowell, Z. Cvetanovic, J. Harris, C.A. Nelson, and C. Ofner. Extending OpenMP for NUMA Machines. In *Proceedings of Supercomputing 2000*, November 2000.

- [14] F. Bodin, P. Beckman, D. Gannon, S. Yang, S. Kesavan, A. Malony, and B. Mohr. Implementing a Parallel C++ Runtime System for Scalable Parallel Systems. In *Proceedings of Supercomputing '93*, November 1993.
- [15] Z. Bozkus, L. Meadows, S. Nakamoto, V. Schuster, and M. Young. PGHPF—An Optimizing High Performance Fortran Compiler for Distributed Memory Machines. *Scientific Programming*, 6(1):29–40, 1997.
- [16] M. Burke and R. Cytron. Interprocedural Dependence Analysis and Parallelization. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, Palo Alto, California, June 1986.
- [17] D. Callahan, Bradford L. Chamberlain, and Hans P. Zima. The Cascade High Productivity Language. In *Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'04)*, pages 52–60, April 2004.
- [18] D. Callahan and K. Kennedy. Compiling Programs for Distributed-Memory Multiprocessors. *Journal of Supercomputing*, 2:151–169, October 1988.
- [19] Bradford L. Chamberlain. *The Design and Implementation of a Region-Based Parallel Programming Language*. PhD thesis, Department of Computer Science and Engineering, University of Washington, 2001.
- [20] Bradford L. Chamberlain, D. Callahan, and H. P. Zima. Parallel Programmability and the Chapel Language. *The International Journal of High Performance Computing Applications*, 2007. Special Issue on High Productivity Programming Languages and Models (in print).
- [21] B. Chapman, P. Mehrotra, and H. Zima. Programming in Vienna Fortran. *Scientific Programming*, 1(1):31–50, Fall 1992.
- [22] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioğlu, Christoph von Praun, and Vivek Sarkar. X10: An Object-oriented Approach to Non-uniform Cluster Computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 519–538, New York, NY, USA, 2005. ACM Press.
- [23] Siddhartha Chatterjee, John R. Gilbert, Robert Schreiber, and Shang-Hua Teng. Automatic array alignment in data-parallel programs. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 16–28, New York, NY, USA, 1993. ACM Press.
- [24] Daniel Chavarría-Miranda, Guohua Jin, and John Mellor-Crummey. Assessing the US Supercomputing Strategy: An Application Study Using IMPACT-3D. In *Proceedings of the 19th International Parallel and Distributed Processing Symposium (IPDPS)*, Denver, CO, April 2005.
- [25] Daniel Chavarría-Miranda and John Mellor-Crummey. An Evaluation of Data-Parallel Compiler Support for Line-Sweep Applications. *The Journal of Instruction-Level Parallelism*, 5, February 2003. (<http://www.jilp.org/vol5>).
- Special issue with selected papers from: The Eleventh International Conference on Parallel Architectures and Compilation Techniques, September 2002. Guest Editors: Erik Altman and Sally McKee.
- [26] A. Choudhary, G. Fox, S. Ranka, S. Hiranandani, K. Kennedy, C. Koelbel, and J. Saltz. Software Support for Irregular and Loosely Synchronous Problems. *International Journal of Computing Systems in Engineering*, 3(2):43–52, 1993. (also available as CRPC-TR92258).
- [27] Peter Christy. Software to Support Massively Parallel Computing on the MasPar MP-1. In *Proceedings of the IEEE Compcon*, 1990.
- [28] Stephanie Coleman and Kathryn S. McKinley. Tile Size Selection Using Cache Organization and Data Layout. In *PLDI '95: Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, pages 279–290, New York, NY, USA, 1995. ACM Press.
- [29] K. D. Cooper, M. W. Hall, K. Kennedy, and L. Torczon. Interprocedural Analysis and Optimization. *Communications in Pure and Applied Mathematics*, 48:947–1003, 1995.
- [30] Cray history. <http://www.cray.com/about-cray/history.html>.
- [31] Cray-1 Computer System. Hardware Reference Manual 224004. Technical report, Cray Research, Inc., November 1977. Revision C.
- [32] David E. Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel Computer Architecture. A Hardware/Software Approach*. Morgan Kaufman Publishers, San Francisco, California, 1999.
- [33] L. Dagum and R. Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *Computational Science and Engineering*, 5(1):46–55, 1998.
- [34] F. Damera-Rogers, D. A. George, V. A. Norton, and G. F. Pfister. VM/EPEX—A VM Environment for Parallel Execution. Technical Report RC 11225(#49161), IBM T. J. & Watson Research Center, Yorktown Heights, New York, January 1985.
- [35] Luiz DeRose and David Padua. A MATLAB to Fortran 90 Translator and Its Effectiveness. In *Proceedings of the 10th International Conference on Supercomputing*, May 1996.
- [36] R. L. Diaconescu and H.P. Zima. An Approach to Data Distributions in Chapel. *The International Journal of High Performance Computing Applications*, 2006. Special Issue on High Productivity Programming Languages and Models (in print).
- [37] M. Flynn. Some Computer Organisations and their Effectiveness. *Trans. Computers*, 21:948–960, 1972.
- [38] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C.-W. Tseng, and M. Wu. The Fortran D Language Specification. Technical Report TR90-141, Department of Computer Science, December 1990.
- [39] Dennis Gannon, William Jalby, and Kyle Gallivan. Strategies for Cache and Local Memory Management by Global Program Transformation. *J. Parallel Distrib. Comput.*,

- 5(5):587–616, 1988.
- [40] Michael Gerndt. *Automatic Parallelization for Distributed-Memory Multiprocessing Systems*. PhD thesis, University of Bonn, Germany, 1989.
- [41] Wolfgang K. Giloi. SUPRENUM: A Trendsetter in Modern Supercomputer Development. *Parallel Computing*, pages 283–296, 1988.
- [42] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitsberg, W. Saphir, and M. Snir. *MPI—The Complete Reference: Volume 2, The MPI Extensions*. Scientific and Engineering Computation Series. MIT Press, Cambridge, Massachusetts, September 1998.
- [43] M. Gupta, S. Midkiff, E. Schonberg, V. Seshadri, D. Shields, K. Wang, W. Ching, and T. Ngo. An HPF Compiler for the IBM SP2. In *Proceedings of Supercomputing '95*, San Diego, California, December 1995.
- [44] Manish Gupta. *Automatic Data Partitioning on Distributed Memory Multicomputers*. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1992.
- [45] L. Hamel, P. Hatcher, and M. Quinn. An Optimizing C* Compiler for a Hypercube Multicomputer. In J. Saltz and P. Mehrotra, editors, *Languages, Compilers, and Run-Time Environments for Distributed Memory Machines*. North-Holland, Amsterdam, The Netherlands, 1992.
- [46] J. Harris, J. Bircsak, M. R. Bolduc, J. A. Diewald, I. Gale, N. Johnson, S. Lee, C. A. Nelson, and C. Offner. Compiling High Performance Fortran for Distributed-Memory Systems. *Digital Technical Journal of Digital Equipment Corporation*, 7(3):5–23, Fall 1995.
- [47] P. Hatcher, A. Lapadula, R. Jones, M. Quinn, and J. Anderson. A Production Quality C* Compiler for Hypercube Machines. In *3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 73–82, April 1991.
- [48] John L. Hennessy and David A. Patterson. *Computer Architecture. A Quantitative Approach*. Morgan Kaufman Publishers, 1996. Second Edition.
- [49] High Performance Fortran Forum. High Performance Fortran Language Specification. *Scientific Programming*, 2(1–2):1–170, 1993. (also available as CRPC-TR92225).
- [50] High Performance Fortran Forum. High Performance Fortran Language Specification, version 1.0. Technical Report CRPC-TR92225, Rice University, Center for Research on Parallel Computation, Houston, Tex., 1993.
- [51] R. Hill. MIMDizer: A New Tool for Parallelization. *Supercomputing Review*, 3(4), April 1990.
- [52] W. Daniel Hillis. *The Connection Machine*. Series in Artificial Intelligence. MIT Press, Cambridge, MA, 1985.
- [53] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiling Fortran D for MIMD Distributed-Memory Machines. *Communications of the ACM*, 35(8):66–80, August 1992.
- [54] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Evaluation of Compiler Optimizations for Fortran D on MIMD Distributed-Memory Machines. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, Washington, DC, July 1992.
- [55] P. Husbands, C. Isbell, and A. Edelman. MATLAB*P: A Tool for Interactive Supercomputing. In *Proceedings of the 9th SIAM Conference on Parallel Processing*. Scientific Computing, 1999.
- [56] ESA/390 Principles of Operation. Technical report, IBM Corporation. IBM Publication No. SA22-7201.
- [57] K. Ikudome, G. Fox, A. Kolawa, and J. Flower. An Automatic and Symbolic Parallelization System for Distributed Memory Parallel Computers. In *Fifth Distributed-Memory Computing Conference*, pages 1105–1114, Charleston, South Carolina, April 1990.
- [58] H. F. Jordan. The Force. In L. H. Jamieson, D. B. Gannon, and R. J. Douglass, editors, *The Characteristics of Parallel Algorithms*. MIT Press, 1987.
- [59] Kendall Square Systems. <http://en.wikipedia.org/wiki/Kendall-Square-Research>.
- [60] Ken Kennedy and Uli Kremer. Automatic Data Layout for Distributed Memory Machines. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(4):869–916, July 1998.
- [61] Jeremy Kepner and Nadya Travinin. Parallel Matlab: The next generation. In *Proceedings of the 2003 Workshop on High Performance Embedded Computing (HPEC03)*, 2003.
- [62] KSR1 Principles of Operation, 1991. Waltham, MA.
- [63] D. Kuck, Y. Muraoka, and S. Chen. On the Number of Operations Simultaneously Executable in Fortran-like Programs and Their Resulting Speedup. *IEEE Transactions on Computers*, C-21(12):1293–1310, December 1972.
- [64] Monica D. Lam, Edward E. Rothberg, and Michael E. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. In *ASPLOS-IV: Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, New York, NY, USA, 1991. ACM Press.
- [65] L. Lamport. The Parallel Execution of DO Loops. *Communications of the ACM*, 17(2):83–93, February 1974.
- [66] B. Leasure. Parallel Processing Model for High-Level Programming Languages. Technical report, American National Standard for Information Processing, April 1994.
- [67] J. Li and M. Chen. Compiling Communication-Efficient Programs for Massively Parallel Machines. *IEEE Trans. Parallel Distrib. Syst.*, 2(3):361–376, 1991.
- [68] J. Li and M. Chen. Generating Explicit Communication from Shared-Memory Program References. In *Proceedings of Supercomputing '90*, pages 865–876, New York, NY, November 1990.
- [69] J. H. Merlin. Adapting Fortran 90 Array Programs for Distributed-Memory Architectures. In H.P. Zima, editor, *First International ACPC Conference*, pages 184–200. Lecture Notes in Computer Science 591, Springer Verlag, Salzburg, Austria, 1991.

- [70] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. *International Journal of Supercomputer Applications and High Performance Computing*, 8(3/4):165–414, 1994. (special issue on MPI, also available electronically via <ftp://www.netlib.org/mpi/mpi-report.ps>).
- [71] Myrias Systems. <http://en.allExperts.com/e/m/myriasis-research-corporation.htm>.
- [72] D. S. Nikolopoulos, T.S. Papatheodorou, C.D. Polychronopoulos, J. Labarta, and E. Ayguade. Is Data Distribution Necessary in OpenMP? In *Proceedings of Supercomputing 2000*, November 2000.
- [73] OpenMP Application Program Interface. Version 2.5. Technical report, OpenMP Architecture Review Board, May 2005. <http://www.openmp.org>.
- [74] Anita Osterhaug. *Guide to Parallel Programming on Sequent Computer Systems*. Prentice Hall, Englewood Cliffs, New Jersey, 1989.
- [75] D. Pase. *MPP Fortran Programming Model*. High Performance Fortran Forum, January 1991.
- [76] M. Quinn, A. Malishevsky, N. Seelam, and Y. Zhao. Preliminary Results from a Parallel MATLAB Compiler. In *Proceedings of the International Parallel Processing Symposium*, pages 81–87, April 1998.
- [77] A. P. Reeves and C.M. Chase. The Paragon Programming Paradigm and Distributed-Memory Multicomputers. In J. Saltz and P. Mehrotra, editors, *Compilers and Runtime Software for Scalable Multiprocessors*. Elsevier, Amsterdam, The Netherlands, 1991.
- [78] A. Rogers and K. Pingali. Process Decomposition Through Locality of Reference. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, Portland, Oregon, June 1989.
- [79] John R. Rose and Guy L. Steele. C*: An Extended C Language for Data Parallel Programming. In *Proceedings Second International Conference on Supercomputing*, volume Vol.II,2-16, 1987. International Supercomputing Institute.
- [80] M. Rosing, R.W. Schnabel, and R.P. Weaver. Expressing Complex Parallel Algorithms in DINO. In *Fourth Conference on Hypercubes, Concurrent Computers, and Applications*, pages 553–560, 1989.
- [81] R. Ruehl and M. Annarato. Parallelization of Fortran Code on Distributed-Memory Parallel Processors. In *International Conference on Supercomputing*, Amsterdam, The Netherlands, June 1990. ACM Press.
- [82] H. Sakagami, H. Murai, Y. Seo, and M. Yokokawa. 14.9 TFLOPS Three-dimensional Fluid Simulation for Fusion Science with HPF on the Earth Simulator. In *SC2002*, November 2002.
- [83] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman. Run-time Scheduling and Execution of Loops on Message-passing Machines. *Journal of Parallel and Distributed Computing*, 8(2):303–312, 1990.
- [84] Vivek Sarkar. Automatic Selection of High Order Transformations in the IBM XL Fortran Compilers. *IBM Journal of Research and Development*, 41(3), May 1997.
- [85] C. L. Seitz. The Cosmic Cube. *Communications of the ACM*, 28(1):22–33, January 1985.
- [86] M. Shimasaki and Hans P. Zima. Special Issue on the Earth Simulator, November 2004.
- [87] D. Slotnick, W. Brock, and R. MacReynolds. The SOLOMON computer. In *Proceedings of the AFIPS Fall Joint Computer Conference*, pages 87–107, 1962.
- [88] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI—The Complete Reference: Volume 1, The MPI Core*. MIT Press, Cambridge, Massachusetts, 2 edition, 1998.
- [89] Lawrence Snyder. *A Programming Guide to ZPL*. MIT Press, Scientific and Engineering Computation Series, March 1999.
- [90] Sun Microsystems, Inc., Burlington, Massachusetts. *The Fortress Language Specification, Version 0.707*, July 2005.
- [91] Thinking Machines Corporation, Cambridge, Massachusetts. *CM Fortran Reference Manual, Version 1.0*, February 1991.
- [92] Transputer: A Programmable Component that Gives Micros a New Name. *Computer Design*, 23:243–244, February 1984.
- [93] R. Triolet, F. Irigoin, and P. Feautrier. Direct Parallelization of CALL Statements. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, Palo Alto, California, June 1986.
- [94] C.-W. Tseng. *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines*. PhD thesis, Rice University, January 1993.
- [95] P. S. Tseng. A Systolic Array Programming Language. In *Fifth Distributed-Memory Computing Conference*, pages 1125–1130, Charleston, South Carolina, April 1990.
- [96] M. Ujaldon, E.L. Zapata, B. Chapman, and H. Zima. Vienna Fortran/HPF Extensions for Sparse and Irregular Problems and Their Compilation. *IEEE Transactions on Parallel and Distributed Systems*, 8(10):1068–1083, October 1997.
- [97] Vax computers. <http://h18000.www1.hp.com/alphaserver/vax/timeline/1986.html>.
- [98] Michael E. Wolf and Monica S. Lam. A Data Locality Optimizing Algorithm. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, pages 30–44, New York, NY, USA, 1991. ACM Press.
- [99] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers*. MIT Press, Cambridge, Massachusetts, 1989.
- [100] H. Zima, H.-J. Bast, and M. Gerndt. SUPERB: A Tool For Semi-Automatic MIMD/SIMD Parallelization. *Parallel Computing*, 6:1–18, 1988.
- [101] H. Zima and B. Chapman. Compiling for Distributed-Memory Systems. *Proceedings of the IEEE*, 81(2):264–287,

February 1993.

- [102] Hans P. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald. Vienna Fortran – A Language Specification. *Internal Report 21, ICASE, NASA Langley Research Center*, March 1992.

The Design and Development of ZPL

Lawrence Snyder

Department of Computer Science and Engineering
Box 352350
University of Washington
Seattle, WA 98195-2350
snyder@cs.washington.edu

Abstract

ZPL is an implicitly parallel programming language, which means all instructions to implement and manage the parallelism are inserted by the compiler. It is the first implicitly parallel language to achieve performance portability, that is, consistent high performance across all (MIMD) parallel platforms. ZPL has been designed from first principles, and is founded on the CTA abstract parallel machine. A key enabler of ZPL's performance portability is its What You See Is What You Get (WYSIWYG) performance model. The paper describes the antecedent research on which ZPL was founded, the design principles used to build it incrementally, and the technical basis for its performance portability. Comparisons with other parallel programming approaches are included.

Categories and Subject Descriptors D.3.2 [Programming Languages]: Language Classifications – Concurrent, distributed and parallel languages; D.3.3 Languages, Constructs and Features – Concurrent programming structures, Control structures, Data types and structures; D.3.4 Processors – Compilers, Retargetable compilers, Optimization, Run-time Environments; C.1.2 Multiple Data Stream Architectures – Multiple-instruction-stream, multiple-data-stream architectures (MIMD), Interconnection architectures; C.4 Performance of Systems – Design studies, Modeling studies, Performance attributes; E.1 Data Structures – Distributed data structures; F.1.2 Modes of Computation – Parallelism and concurrency.

General Terms: Performance, Design, Experimentation, Languages.

Keywords: performance portability; type architecture; parallel language design; regions; WYSIWYG performance model; CTA; problem space promotion.

1. Starting Out

On October 7, 1992 Calvin Lin clicked on an overhead

Permission to make digital/hard copy of part of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date of appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Permission may be requested from the Publications Dept., ACM, Inc., 2 Penn Plaza, New York, NY 11201-0701, USA, fax:+1(212) 869-0481, permissions@acm.org

©2007 ACM 978-1-59593-766-7/2007/06-ART8 \$5.00
DOI 10.1145/1238844.1238852

<http://doi.acm.org/10.1145/1238844.1238852>

projector in a Sieg Hall classroom on the University of Washington campus in Seattle, and began to defend his doctoral dissertation, *The Portability of Parallel Programs Across MIMD Computers*. The dissertation, which was a feasibility study for a different approach to parallel programming, reported very promising results. At the celebratory lunch at the UW Faculty Club we agreed that it was time to apply those results to create a new language. We had studied the critical issues long enough; it was time to act. At 9:00 AM the following morning he and I met to discuss the nature of the new language, and with that discussion the ZPL parallel programming language was born.

It would be satisfying to report that that morning meeting was the first of a coherent series of events, each a logical descendant of its predecessors derived through a process of careful research and innovative design, culminating in an effective parallel language. But it is not to be. ZPL is effective, being a concise array language for fast, portable parallel programs, but the path to the goal was anything but direct. That we took a meandering route to the destination is hardly unique to the ZPL Project. We zig-zagged for the same reason other projects have: language design is as much art as it is science.

What was unique to our experience was designing ZPL while most of the parallel language community was working cooperatively on High Performance Fortran (HPF), a different language with similar goals. On the one side was our small, coherent team of faculty and grad students, never more than ten, designing a language from scratch. On the other side was a distributed, community-wide effort with hundreds of participants, generous federal funding, and corporate buy-in dedicated to parallelizing Fortran 90. It had all of the features of a David and Goliath battle. Language design is much more complex and subtle than combat, however, so “battle” mischaracterizes the nature of the competition, as I will explain.

In the following I describe the design and development of ZPL. One objective is to explain the important technical decisions we made and why. Their importance derives both from their contributions to ZPL's success and their potential application in new (parallel) languages. The other objective is to convey the organic process by which ZPL was created, the excitement, the frustrations, the missteps, the epiphanies. Language design for us was a social activity, and just as a software system's structure often reflects the organization of the group that created it, the ZPL team pressed its unique profile into the resulting language.

To follow the winding path, it is necessary to know the intended destination.

2. Goals

By the morning of that first meeting the goals were well established. They were even established before Lin started his dissertation, because the whole ZPL effort was preceded by a decade of parallel programming research. Our earlier projects had created two other parallel languages and a parallel programming environment, tested them and found them wanting. Though they failed to achieve the design goals, each effort let us explore the design space and test the efficacy of different approaches. Lin’s dissertation was the transitional research moving us from exploring the problem to solving it. All of these attempts shared the same objective: To be a practical language that delivers performance, portability and convenience for programming large scientific and engineering computations for parallel computers.

2.1 Performance

The performance goal is self-evident because the main reason to program a parallel computer is to complete a computation faster than with a sequential solution. A parallel programming language that cannot *consistently* deliver performance will not be used. Consistency is essential because programmers will not spend effort programming on the *chance* that the code runs fast. It must be a certainty. Further, the degree to which the parallel solution outperforms the sequential solution is also important, since it is a measure of the programmer’s success. Because of the naïve assumption that P processors should speed up a computation by a factor of P , users expect their programs to approach that performance. In consequence language designers must repress any temptation to “spend” portions of that factor-of- P potential for language facilities. A common example is to introduce $\log P$ overhead to implement the shared memory abstraction [1]. Our performance *goal* was to deliver the whole potential to the programmer.

2.2 Portability

Program portability, meaning that programs “perform equivalently well across all platforms,” was the second property we wanted to achieve.¹ Portability, once a matter of serious concern, had not been a significant issue in programming language design for sequential computers since the development of instruction set architectures (ISAs) in the 1960s. Sequential computer implementations are extremely similar in their operational behavior and any programming language more abstract than assembly language could be effectively compiled to 3-address code with a stack/heap memory model, and then translated to the ISA of any sequential computer.

By contrast, parallel computer architecture had not (has not) converged to a single machine model. Indeed, when the ZPL effort began, large **SIMD** (single instruction stream, multiple data stream) computers were still on the

market. As it was obvious from the beginning that SIMD machines are too limited to support general purpose computing, our focus has always been on **MIMD** (multiple instruction stream, multiple data stream) machines. Among MIMD computers, though, dozens of different architectures have been built and many more proposed during the period of our programming language research [2]. Compiling efficient code for all of them is a challenge.

Portability was an issue because, as one researcher from Lawrence Livermore National Laboratory (LLNL) lamented at a Salishan meeting² at the time, “Before we get the program written they roll in a new computer, and we have to start from scratch.” The problem, as in the 1950s, was that programmers tried to achieve performance by exploiting specific features peculiar to the architecture, and computer vendors encouraged them to do so to secure their market advantage. For example, the popular-at-the-time hypercube communication network was often explicitly used in programs [3]. Once the topology was embedded into the code, the program either had to be rewritten or considerable software overhead had to be accepted to emulate a hypercube when the next machine “rolled in” with a different topology. This problem was extremely serious for the national labs, accustomed as they were to running programs for years, the so-called “dusty decks.”

Achieving portability in parallel language design entails resolving a tension: Raising the level of abstraction tends to improve portability, while programming at a lower level, closer to the machine, tends to improve performance. Raising the level high enough to overtop all architectural features, however, may penalize performance to an unacceptable degree if layers of software are needed to map the abstractions to the hardware. The purpose of Lin’s dissertation research was to validate the portability solution we intended to use in ZPL (the CTA approach, see below).

As a postscript, portability as used here, meaning portability with (parallel) performance, was not much discussed in the literature as the Lin work was being published. I think we were the only group worried about it. Soon, it became widely mentioned. It was never clear to me that the researchers who claimed portability for their languages understood the tension just described. Rather, I believe they interpreted portability in what I began thinking of as the “computability sense.” Computability’s universality theorem says any program can run on any modern computer by (some amount of) emulation. This fact applies to parallel programs on parallel computers. All parallel languages produce portable programs in this sense. Our portability goal required a program in language X, which delivered good performance on platform A, quantified, say, as 80% of the performance of a custom, hand-coded program for A, to deliver roughly the same good performance when compiled and run on platform B. This is the sort of portability sequential programs enjoy, and users expect it of parallel programs, too. Portability in the “computability sense” says nothing about maintaining performance.

¹ “Portability” has a range of meanings from “binary compatible” to “executable after source recompilation;” the latter applies here.

² The national labs have regularly held a meeting of parallel computation experts at the Salishan Conference Center on the Oregon Coast.

2.3 Convenience

Convenience, meaning ease of use, was the third goal. Inventing a language that is easy to use is a delicate matter. To aid programmers, it is important to match the language's abstractions to the primitive operations they use to formulate their computations. When we began (and it remains true) there were few "standard" parallel language constructs. What statement form do we use to express a pipelined data structure traversal? What is standard syntax for writing a (custom) parallel prefix computation? It is necessary to create new statement forms, but doing so is subtle. They must be abstract, but still give programmers effective control. Of course, picking syntax for them is guaranteed to create controversy.

On the other side of the convenience equation, the problem domain influences the choice of language facilities. Because high-end applications tend to be engineering and scientific computations, good support for arrays, floating-point computations and libraries was essential. Additionally, keeping the memory footprint contained was also essential, because high-end computations are as often memory bound than computation bound.

2.4 Additional Constraints

Our plan was to build ZPL from scratch rather than extending an existing (sequential) language. Our two previous language designs included an instance of each approach. Many benefits have been cited for extending an existing language, such as a body of extant programs, knowledgeable programmers, libraries, etc. It is an odd argument, since these resources are all sequential. To exploit them implies either that users are content to run sequential programs on a parallel computer, or that there is some technique to transform existing sequential programs into parallel programs; if that is possible, why extend the language at all?

While on the topic of techniques to transform sequential programs, notice that the "parallelizing compiler" approach is the ultimate solution to the *practical* parallel programming problem: It produces—effortlessly from the programmer's viewpoint—a parallel program from a sequential program. It seems miraculous, but then compilers are pretty amazing anyhow, so it is easy to see why the approach is so often pursued. I first argued in 1980 [4] that this technique would not work based on two facts: (1) compilers, for all their apparent magic, simply transform a computation from a higher to a lower level of abstraction without changing the computational steps significantly; (2) solving a problem in parallel generally requires a paradigm shift from the sequential solution, i.e. the *algorithm* changes. Despite considerably more progress than I might have expected, the prospects of turning the task over to a compiler remain extremely remote. ZPL was focused on writing *parallel* programs (as were all of its ancestors).

Our experience extending an existing language (Orca C, discussed below) demonstrated the all-too-obvious flaw in trying extend a sequential language: With an existing language, programmers familiar with it expect its statements and expressions to work in the usual way. How can this expectation be managed? *Preserving* the sequential semantics while mapping to a parallel context is difficult to impossible, especially considering that performance is the goal. *Changing* the semantics to morph them into a parallel

context is annoying to infuriating for programmers, especially considering the potential for introducing subtle bugs; further, it renders the existing program base useless. Having done it once, I was not about to repeat the mistake. So, we started from scratch—we called it *designing from first principles*. This also has the advantage that the needs of the programmer and the compiler can be integrated into the new language structures.

Finally, we decided to let parallelism drive the design effort, avoiding object orientation and other useful or fashionable, but orthogonal, language features. If we got the parallelism right, they could be added. Before that point they would be distracting.

3. Overview of the ZPL Project

To build a superstructure around ZPL to allow direct access to the interesting parts of the research, we take a rapid tour through nearly a dozen years of the project.

3.1 The Time Line

Prior to the October 1992 meeting, several efforts in addition to Lin's dissertation produced foundations on which we built. One of these, the CTA Type Architecture, is especially critical to the language's success and will be described in detail in a later section.

Beginning in October, Lin and I worked daily sketching a language based on his dissertation work. By March, 1993, the language was far enough along that he could present it at the final meeting of the Winter Quarter's CSE590-O, *Parallel Programming Environments Seminar*. His hour-long lecture was intended to attract students to the Spring Quarter CSE590-O, which we planned to make into a language design seminar. (The language was not yet named.) We would program using the language's features, as well as try to extend its capabilities. The advertisement was successful. The project began that March with seminar students Ruth Anderson, Brad Chamberlain, Sung-Eun Choi, George Foreman, Ton Ngo, Kurt Partridge and Derrick Weathersby. A few terms later, E Chris Lewis joined to be the final project founder. See Figure 1.

As the seminar students learned the base language and began contributing to the design in Spring 1993, names were frequently suggested, usually humorously. Lin had called it Stipple in his presentation, because he had used stipple patterns liberally in his diagrams. I had called it Zippy, suggesting it produced fast programs, but that was too informal. Eventually, we gravitated to ZPL, short for Z-level Programming Language, as explained below.

With performance as the first goal, we adopted Project Policy #1: Only include in the language those features that we know how to compile into efficient code. At this point our main guidance on what could be done efficiently was Lin's dissertation. So, the language contained very little at the start. Indeed, when the design was first described in 1993 [5], ZPL was called "an array sublanguage" because so many features were missing. There was enough expressiveness, however, to cover common inner loop computations. For example, the 4-nearest neighbor stencil iteration could be expressed with these lines (plus declarations),

```
[1..n,1..n] repeat
    Temp := (A@north + A@east
              + A@west + A@south)/4;
    err := max<<abs(Temp-A);
```



Figure 1. Snapshot from a ZPL Design Meeting, 1995. Standing, from left, Jason Secosky, E Chris Lewis, Larry Snyder, Ton Ngo, Derrick Weathersby and Judy Watson; seated, Sung-Eun Choi, Brad Chamberlain, Calvin Lin

```
A := Temp;
until err < tolerance;
```

The new programming abstractions introduced by ZPL at this point were *regions* (the indices in brackets), *directions* (the @-constructs), and the *global-view distributed memory model*, as explained below.

As we designed we applied the general heuristic of optimizing for the common case. This would often have a profound effect on the language. Most frequently it caused us to decide *not* to adopt a too-general approach; we would table the final decision pending further inspiration.

Compiler construction began by summer 1993. Project Policy #2: The compiler must always create fast and portable code, starting at the beginning. The next year, the performance and portability results reported for ZPL [6] showed that our research goals could be achieved, at least for the sublanguage. The policy had, at least, been instantiated properly. Nearly all of the ZPL papers show performance results on several different parallel architectures.

Project meetings and implementation started out in the Blue CHiP Lab in Sieg Hall, but with the Computer Science and Engineering department critically short on space, we gave up the lab after the summer of 1994. The team never had another lab, despite the tremendous benefits of having a focal point for a research project. The grad students were distributed across several offices, most of them in a temporary manufactured building next to Sieg Hall known to the team as Le Chateau. With everyone scattered, implementation was coordinated by network, and the de-

sign discussions were conducted in conference rooms. In my opinion the absence of lab space slowed progress to a noticeable degree.

During 1995-6 the team extended ZPL's expressiveness substantially, and our few users gave us feedback on the design. Indeed, in accordance with Project Policy #3: Language design proceeded with a handful of representative operations or computations drawn from user applications. By this we hoped to match the needs of the problem domains. By 1996 a succinct, high-level stand-alone language had been designed and implemented —dubbed ZPL Classic by the designers.

Comparisons with standard benchmarks, for example, the SPLASH and xHPF suites, showed that the ZPL compiler was doing well on such parallel-specific aspects as communication performance, but that its scalar performance was only average. For these well-worked benchmark codes, programmers could incorporate hand optimizations into their low-level message passing programs to make them extremely efficient. In high-level languages like ZPL it is the compiler that is responsible for such optimizations because programmers do not write the low-level code like array-traversals loops. So, after an additional year devoted mostly to improving ZPL's scalar performance, the compiler was officially released in July 1997.

The team changed slowly over time. Jason Secosky joined in 1994. Lin joined the faculty at UT Austin in 1996. Ruth Anderson, George Forman and Kurt Partridge moved on to other dissertation topics. Taylor van Vleet, Wayne Wong and Vassily Litvinov joined the project, con-

tributed and moved on. Ton Ngo returned to IBM in 1996. In 1997 Maria Gulickson, Douglas Low and Steven Deitz joined as project members. Periodically, researchers from client fields, such as chemistry, astronomy and mechanical engineering, would join our group meetings.

One effect of Policy #1, the “add no feature until it compiles to fast code” policy, was to make ZPL Classic a rather static language that emphasized creating structures through declarations rather than building them on the fly. But creating ZPL Classic had also given us sufficient insight to be more dynamic. So, following the completion of ZPL Classic we began work on Advanced ZPL (A-ZPL) to enhance programmer flexibility. New features included built-in sparse arrays, upgrading regions and directions to first-class status, and introducing facilities for dynamically controlling data distributions and processor assignments. The resulting language was much more flexible.

Users have programmed in ZPL since 1995. ZPL programs are much more succinct than programs written in other general approaches. Experiments show that ZPL programs consistently perform well and scale well. On the NAS Parallel Benchmarks ZPL nearly matches performance against the hand-optimized MPI with Fortran or C; on user-written programs, ZPL is generally significantly faster than MPI programs. (See below.) Our ZPL-to-C compiler emits code that can be compiled for any parallel machine and typically requires little tuning. The ZPL compiler has been available open source since 2004. Concepts and approaches from ZPL have been incorporated into Cray’s Chapel, IBM’s X10 and other parallel languages.

4. Contributions

Though they are described in detail below, it may be helpful to list the main contributions of the ZPL project now as a link among various threads of the story.

```

program Life;
  /* In a toroidal world externally initialized, organisms with 2 or 3
   neighbors survive to the next generation; empty cells with 3
   neighbors give birth in next generation; all others die */

  config const n : integer = 16;
  region R = [1..n, 1..n];
  direction
    nw = [-1, -1]; no = [-1, 0]; ne = [-1, 1];
    w = [0, -1]; e = [0, 1];
    sw = [1, -1]; so = [1, 0]; se = [1, 1];
  var
    TW : [R] boolean;
    NN : [R] byte;

  procedure Life();
  begin
    -- Initialize the world here
    [R] repeat
      NN := TW@^nw + TW@^no + TW@^ne
        + TW@^w + TW@^e
        + TW@^sw + TW@^so + TW@^se;
      TW := (TW & NN = 2) | (NN = 3);
    until !(|<< TW);
  end;

```

Conway’s Life
The world is $n \times n$; default to 16; change on command line.
Index set of computation; n^2 elements.
Reference vectors for 8 nearest neighbors ...
Declarations for two array variables both $n \times n$.
Problem state, The World
Work array, Number of Neighbors
Entry point procedure; same name as program.
Region R means all operations apply to all indices.
Add eight nearest neighbor bits (type coercion follows C)
Caret(\wedge) means neighbor is a toroidal reference at array edge.
Update world with next generation.
Continue till all die out; use an OR reduction.

Figure 2. Conway’s Game of Life in ZPL. The problem state is represented by the $n \times n$ Boolean array TW . On each iteration of the repeat loop a generation is computed: the first statement counts for each grid position the number of nearest neighbors, and the second statement applies the “rules”; the iteration ends when no more cells are alive.

I believe the principal contributions are

- Basing a parallel language and its compiler on the CTA machine model (type architecture).
- Creating a global view program language for MIMD parallel computers in which parallelism is inherent in the operations’ semantics.
- Developing the first parallel language whose programs are fast on all MIMD parallel computers.
- Designing from first principles parallel abstractions to replace shared memory.
- Creating new parallel language facilities including regions, flooding, remap, mscan and others, all described below.
- Applying various design methodologies, including an always fast and portable implementation, designing from user problems, evolutionary design, etc.
- Creating the problem space promotion parallel programming technique.
- Inventing numerous compilation techniques and compiler optimization approaches including the Ironman communication layer, loop-fusion and array contraction algorithms, the factor-join technique, and others.

There are many other smaller contributions, but these successes most accurately characterize the advances embodied in ZPL.

5. The Programmer’s View of ZPL

To illustrate ZPL programming style in preparation for a careful explanation of its technical details, consider writing a program for Conway’s Game of Life. Over the years we used various introductory examples, but we eventually adopted the widely known Game of Life and joked that it solved biologically inspired simulation computations.

5.1 Life As A ZPL Program

Recall that the Game of Life models birth and death processes on a two dimensional grid. The grid is toroidally connected, meaning that the right neighbors of the right edge positions are the corresponding positions on the left edge; similarly for the top and bottom edges. The game begins with an initial configuration, the 0th generation, in which some grid positions are occupied by cells. The $i+1$ st generation is found from the i th generation by eliminating all cells with fewer than two neighbors or more than three; additionally, any vacant cell with exactly three neighbors has a cell in the $i+1$ st generation. See Figure 2 for the ZPL program simulating Life.

The following ZPL concepts are illustrated by the Life program.

The **config const** is a declaration setting **n**, the length of the edge of the world, to 16. Configuration constants and variables are given default values in their declarations that can be changed on the command line later when the program runs.

Regions, which are new to ZPL, are like arrays without data. A **region** is an index set, and can take several forms. The **index range** form used in Life,

```
region R = [1..n, 1..n];
```

declares the lower and upper index limits (and an optional stride) for each dimension. (Any number of dimensions is allowed.) In this example, R is the set of n^2 indices {(1,1), (1,2), (1,3), ..., (n,n)}.

The next declaration statements in the Life program define a set of eight directions pointing in the cardinal directions. A **direction** is a vector pointing in index space. For example, **no** = [-1, 0] is a direction that for any index (i,j) refers to the index to its north (above), i.e. $(i-1, j)$. Directions, another concept introduced in ZPL, provide a symbolic way to refer to near neighbors.

Regions are used in two ways. In

```
var
  TW : [R] boolean;
  NN : [R] byte;
```

the region ([R]) provides the indices for declaring arrays. So, the world (TW) is an n^2 array of Boolean values.

The other use of regions is to control the extent of array computations. For example, by prefixing the **repeat**-statement with [R] we specify that all array computations in the statement are to be performed on indices from R. That is, the operations of the two assignment statements in the body of the **repeat** and the loop condition test, all of which involve expressions whose operands are arrays, will be performed on the n^2 values whose indices are given in the region R. In this way programmers have global (declarative) control over which parts of arrays are operated upon.

The computation starts by generating or inputting the initial configuration. Within the **repeat**-loop the first statement

```
NN := TW@^nw + TW@^no + TW@^ne
      + TW@^w           + TW@^e
      + TW@^sw + TW@^so + TW@^se;
```

counts the number of neighbors by type-casting Booleans to integers. This is an array operation applying to all elements in the region R. Therefore **TW@^no**, for example,

refers to the *array* of elements north of the elements in R; @ⁿ is verbalized **wrap at**. The indices for these north neighbors are found by adding the direction vector to the elements of R, i.e. $R + [-1, 0]$. The translated indices (with wraparound) reference an $n \times n$ array of “north neighbors,” and similarly, for the other direction-modified addition operands. The additions are performed element-wise, producing an $n \times n$ result array whose indices are in R. The result, the count of neighbors, is assigned to NN.

The next statement

```
TW := (TW & NN = 2) | (NN = 3);
```

applies the Game of Life rules to create the TW Boolean array for the next generation. Thus, any position where a cell exists in the current generation and has two neighbors, or the position has three neighbors, becomes true in the next generation. As before, these are array operations applying to all elements of R as specified by the enclosing region scope.

Finally, the loop control

```
until !(|<< TW);
```

performs an OR-reduction ($|<<$) over the new generation; that is, it combines the elements of TW using the OR-operation. If the outcome is not true, no cells exist and the repetition stops.

It is a small computation and only requires a small ZPL program to specify.

5.2 Parallelism in Life

Although the Life program contains some unfamiliar syntax and several new concepts, it is composed mostly of routine programming constructs: declarations, standard types, assignment statements, expressions, control structures, etc. Indeed, it doesn’t look to be parallel at all, since the familiar concepts retain their standard meanings: declarations state properties without specifying computation, statements are executed one at a time in sequence, etc.

The Life program is **data parallel**, meaning that the concurrency is embodied in the array operations. For example, when the eight operand-arrays are added element-wise to create the value to be assigned to NN, the order of evaluation for the implementing scalar operations is not specified, and can be executed in parallel.

ZPL programmers know much more about the program’s parallelism than that. Using ZPL’s WYSIWYG Performance Model (explained below), programmers have complete information about how the program is executed in parallel: They know how the compiler will allocate the data to the processors, they know a small amount of point-to-point communication will be performed at the start of each loop iteration, they know that operations of each statement will be fully concurrent, and that the termination test will require a combination of local parallel computation followed by tournament tree and broadcast communication. (Details are given below.) Though programmers do not write the code, they know what the compiler will produce.

6. Antecedents

To set the context for the ZPL development, recall that in the 1980s and early 1990s views on how to program parallel machines partitioned into two categories that can be dubbed: No Work and Much Work. The No Work commu-

nity expected compilers to do all of the parallelization. Most adherents of this view believed a sequential (Fortran) program was sufficient specification of the computation, although a sizeable minority believed one of the more “modern” functional or logic or other languages was preferable. The Much Work community believed in programming to a specific machine, exploiting its unique features. Their papers were titled “The c Computation on the d Computer,” for various values of c and d , much as technical papers had been in the early days of computing. And conference series such as the Hypercube Conference were devoted to computations on parallel computers defined by their communication topology (binary n -cube). Given the state of the art, the No Work community could have portability but little prospect for performance, while the Much Work community could have performance but little prospect of portability. One without the other is useless in practice. We thought we knew how we might get both, but we were exploring new territory.

6.1 The Importance of Machines

I take as a fundamental principle that programming languages designed to deliver performance should be founded on a realistic machine model [1]. Since the main point of parallel programming is performance, it follows that the principle applies to all parallel languages. I didn’t always know or believe the principle; it emerged in work on ZPL’s language antecedents. The first of these was the Poker Parallel Programming Environment, a product of the Blue CHiP Project and my first real attempt at supporting parallel programming [7].

The Blue CHiP Research Project designed and developed the Configurable, Highly Parallel (CHiP) computer [8], a MIMD parallel machine with programmable interconnect like FPGAs of today. That is, the processors are not permanently connected to each other in a fixed topology; the programmer specifies how the processors are to connect. The programmer designing a “mesh algorithm” connects the processors into a mesh topology, and when the next phase requires a tree interconnect the processors are reconfigured into a tree. The CHiP machine was my answer to the question, “How might VLSI technology change computers?”

While in the throes of machine design in January 1982, I organized a team of a dozen grad students and faculty to begin work on its programming support. Poker was a first attempt at an Interactive Development Environment (IDE) for parallel programming; it was intended to support every aspect of CHiP computer programming.³ Because the CHiP

computer is MIMD, Poker provided the ability to write code for each processor; because the CHiP computer is configurable, it provided the ability to program how the processors should be interconnected; because parallel programming was seen as occurring in phases, it provided the ability to orchestrate the whole computation, globally.

Though we thought of it as a general-purpose programming facility, Poker was actually tightly bound to the underlying CHiP architecture, a 2D planar structure designed more to exploit VLSI than to provide a general-purpose parallel computing facility. Poker was perfectly matched to computations that could run efficiently on the CHiP platform, that is, VLSI-friendly computations such as systolic algorithms and signal processing computations. But the language became more difficult to use in proportion to how misaligned the computation was with the CHiP architecture. The tight binding between the CHiP machine and the programming language limited Poker’s generality, but the successful cases clearly illustrated the value of knowing the target machine while programming. This was an enduring contribution of the Poker research: Programmers produce efficient code when they know the target computer, but directly exploiting machine features embeds assumptions about how the machine works and can impose unnecessary requirements on the solution.

If not the CHiP architecture, then what? At the other end of the spectrum from specific architectures are the abstract concurrent execution environments used for functional languages, logic languages, PRAM computations, etc., all of which were extremely popular in the 1980s [9-11]. These very abstract “machines” are powerful and flexible; they do not constrain the programmer and are easy to use. But how are they implemented efficiently? They did not (and do not) correspond to any physical machine, and emulating them requires considerable overhead. It was evident that fundamental issues concerning memory reference limit the feasibility of all of these approaches. A solution might one day be found, but in the meantime the practical conclusion I came to was: It is just as easy to be too general about the execution engine as it is to be too specific.

6.2 Type Architecture

The sum of what I thought I knew in the summer of 1985 could be expressed in two apparently contradictory statements:

- A tight binding between a programming facility and a machine leads to efficient programs, but by using it programmers embed unnecessary assumptions into the solution.
- A weak or non-existent binding between a programming facility and a physical machine leads to easy-to-write computations that cannot be efficiently implemented.

³ Of course, there were no IDEs to build on at the time, but the Alto work at Xerox PARC [65] illustrated a powerful interactive graphic approach, which we did try to borrow from with Poker. We designed Poker around a bit-mapped display attached to a VAX 11/780, our proto-workstation. Since another VAX 11/780 supported the entire Purdue CS department, our profligate use of computer power to service a single user was deemed ludicrous by my faculty colleagues. It is quaint today to think that dedicating a 0.6 MIPS computer with 256K RAM is wasteful; the surprising aspect of it was how quickly (perhaps 3-4 years) it became completely reasonable. Our Poker system used windows and interactive graphics, but we didn’t understand the windows technology well enough. The resulting

system was very cumbersome and fragile. Trying and failing to make a technology work gives one a deep appreciation and respect for those who do get it right [66].

I decided to explain these two “facts” in an invited paper that eventually appeared under the title *Type Architecture, Shared Memory and the Corollary of Modest Potential* [1]. The paper was purposely provocative at the request of the *Annual Reviews* editors. Accordingly, rather than presenting the two “facts” directly, it argued that parallel models based on a flat, shared memory—the usual abstraction when implementation is ignored and the assumption in the functional/logic/PRAM world—are best avoided on the grounds of poor practical parallel performance. (The point was made more rigorously sometime later [12].) The paper went on to present an alternative, the CTA Type Architecture.

A **type architecture** is a machine model abstracting the performance-critical behavior of a family of physical machines. The term “type” was motivated by the idea of a “type species” in biology, the representative species that exhibits the characteristics of a family, as in the frog *Rana rana* characterizing the family Ranidae. Alas, given how overused the word “type” is in computer science, it was an easily misinterpreted name.

Machine models have been common since Turing introduced his machine in 1936 [13]. But type architectures are not just machine models. They are further constrained to describe accurately the key performance-critical features of the machines, usually in an operational way. For example, in the familiar Random Access Machine Model (*a.k.a.* the von Neumann machine), there is a processor, capable of executing one instruction at a time, and a flat memory, capable of having a randomly chosen, fixed-size unit of its storage referenced by the processor in unit time; moreover, instructions are executed in sequence one per unit time. These characteristics impose performance constraints on how the RAM model can work that inform how we program sequential computers for performance. (Backus described some of these features as the von Neumann Bottleneck [14], and used them to motivate functional languages.) Notice that most features of physical machines are ignored —instruction set, registers, etc.—because they affect performance negligibly.

The idea of a type architecture derived from several months’ reflection on how we write efficient sequential programs to be used as an analogy for how we should write efficient portable parallel programs. The result of this thought-experiment was essentially making explicit what we all understand implicitly: Usually, programmers don’t know what their target machine will be and it changes frequently, yet they are perfectly effective writing fast programs in languages like C. Why? Because

- they understand the performance constraints of the RAM model,
- they understand generally how C maps their code to the model, and
- when the program runs on real hardware it performs as the model predicts.

What is essential about the RAM model is that it gives a true and accurate statement of what matters and what doesn’t for performance. For example, programmers prefer binary search to sequential search because random access is allowed, and 1-per-unit-time data reference and 1-per-unit-

time instruction execution implies sequential search is slower when the list is longer than several elements. It’s not enough to give a machine model; Turing gave a machine model. It’s essential that the model accurately describe the performance that the implementing family of machines can deliver. A type architecture does that.

Notice that Schwartz’s idealistic but otherwise brilliant “Ultracomputer” paper [15] had tried to ensure some degree of realism in a parallel model of computation. His work influenced my thinking, though he was more concerned with algorithmic bounds than production parallel programs, and he was prescriptive rather than descriptive.

So, in the same way that the RAM model abstracts sequential machines and is the basis for programming languages like C, enabling programmers to know the performance implications of their decisions and to produce portable and efficient sequential programs, a type architecture abstracting the class of MIMD parallel machines could be used to define a language enabling programmers to write code that runs well on all machines of that type. MIMD parallel machines, unlike sequential computers, exhibit tremendous variability. What is their type architecture like?

6.3 CTA – Candidate Type Architecture

Abstracting the key characteristics of MIMD parallel machines was somewhat easier than one might have expected given the wide variability of real or proposed architectures. Clearly, it has some number, P , of processors. What’s a processor? Because the parallel machines are MIMD, the processors each have a program counter, program and data memory, and the ability to execute basic instructions; so the RAM type architecture is a convenient (and accurate) definition of a processor. This gives the CTA the attractive property that a degenerate parallel computer, that is, $P = 1$, is a sequential computer, establishing a natural relationship between the two.

Besides processors, the other dominant characteristic of parallel machines is the communication structure connecting the processors. Because connecting every pair of processors with a crossbar is prohibitively expensive, parallel architectures give up on a complete graph interconnection and opt for a sparser topology. I had expected that the parallel type architecture would specify some standard topol-

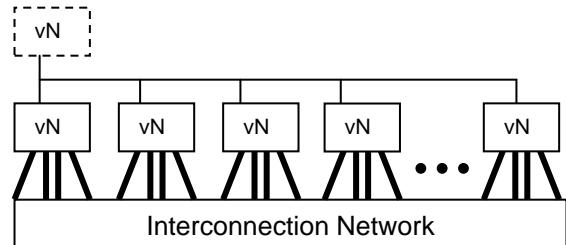


Figure 3. CTA Type Architecture Schematic. The CTA is composed of P von Neumann processors each with its own (program and data) memory, program counter and d-way connection to the communication network; the dashed processor is the “global controller” (often only logical) with a thin connection to all machines; not shown is the key constraint that on-processor memory reference is unit time; off-processor memory reference is λ time.

ogy [16] such as the shuffle-exchange graph or perhaps a 3D torus, but with so many complex issues surrounding interprocessor communication, I was uncertain as to which topology to pick. So, I decided to leave the topology unspecified, and call the machine the *Candidate Type Architecture*, pending a final decision on topology and possibly other features. In retrospect this was an extremely lucky decision, because I eventually decided topology is not any more important to parallel programmers than bus arbitration protocols are to sequential computer programmers. (Also, leaving topology unspecified prevents programmers from embedding topological assumptions in their code, as was criticized under the Antecedents discussion above.) What does matter to programmers, and what was specified with the CTA, is the performance implications of whatever topology the architect ultimately chooses.

The key performance implication of the communication structure is that there is a large latency for communicating between any pair of processors. Though there is also a modest bandwidth limit, it is latency that affects performance—and therefore programming—the most. **Latency**, eventually quantified as λ , cannot be explicitly given as a number for a long list of obvious reasons even for a specific topology: Costs grow with P ; the paths connecting different pairs of processors have different lengths in most topologies; certain routing strategies have no worst-case delivery time [17]; there is network congestion and other conflicts; costs are greatly affected by technology and engineering, etc. But, it is a fact that compared to a local memory reference—taken to be the *unit* of measurement—the latency of a reference to another processor’s memory is very large, typically 2-5 orders of magnitude. The λ value has a profound effect on parallel programming languages and the parallel programming enterprise. The number is so significant that the actual details of the communication network don’t matter that much: If a program is well written to accommodate λ in this range, then it will be well written for a value 10 times as large or one tenth as large.

There are a few other conditions on the CTA: The interconnection network must be a low-degree, sparse topology if it is to be practical to build machines of interesting sizes. A distinguished processor known as the controller has a narrow connection to all processors, suitable for such things as **eurekas** in which one processor announces to all others. Though largely deprecated in many programmers’ minds because processor 0 can act as the controller, various parallel computers have had controllers, and I think it captures an important capability. Notice also that the CTA does not specify whether a parallel machine has hardware support for shared memory. Not specifying shared memory support is consistent with the family of computers being abstracted: some have it and some don’t. Overall, the CTA is a simple machine, as shown in Figure 3.

The CTA, despite its designation as a *candidate* type architecture, has remained unchanged since its original specification 20 years ago, and it has been useful. It provided the basis for demonstrating the inherent weakness in shared memory programming abstractions [12]. It is the machine model used for ZPL. It is the machine model message passing programmers use. It is the machine of the LogP model [18]. Further, the type architecture approach—but not the CTA—has been used for sequential computers with FPGA attached processors [19].

6.4 Preparatory Experiments

From 1982, the genesis of Poker, to 1992, the genesis of ZPL, the Blue CHiP project expended considerable effort trying to solve the parallel programming problem. We tried to enhance Poker. We designed and implemented a language (SPOT) based on the lessons of Poker, but it was unsatisfactory. After creating the CTA, we designed and implemented a language (Orca-C) based on it, but it too was unsatisfactory. Realizing how difficult the problem is, we decided to run a series of experiments (Lin’s feasibility studies) to test out the next language (ZPL) before implementing it. This section skips all of the details, focusing only on the work leading up to Lin’s dissertation.

One derivative from the Poker work was a classification of **parallel programming levels**. Recall a description from three sections back:

Because the CHiP computer is MIMD, Poker provided the ability to write code for each processor; because the CHiP computer is configurable, it provided the ability to program how the processors should be interconnected; because parallel programming was seen as occurring in phases, it provided the ability to orchestrate the whole computation, globally.

These three aspects of Poker programming were eventually labeled **X**, **Y**, **Z**, where X referred to programming processors, Y referred to programming the communication, and Z referred to programming the overall orchestration of the computation. (These were temporary labels pending better names, which were never chosen.) The classification influenced our thinking about programming, and the top level, Z, was particularly important, being the global view of the computation. Z gave ZPL its name: **Z-level Programming Language**.

Two language efforts contributed to a maturing of our ideas. The first was SPOT [20], an effort to raise the level of abstraction of Poker while remaining tightly bound to a CHiP-like architecture. SPOT introduced array abstractions, but mostly it contributed to the recognition, explained above, that coupling too tightly to a specific machine is too constraining. The second effort, known as Orca-C, was an attempt to extend C to a CTA-based model of computation.⁴ The programming was processor-centric, i.e. very much X-level. A prototype was produced, and several application programs, including the Simple benchmark, were written and run on parallel hardware [21]. Being astonished to discover that more than half of the Simple code was devoted to handling boundary conditions, we committed ourselves to attending to the “edges” of a computation in future abstractions. Though Orca-C gave us experience designing and implementing a CTA language and helping users apply it, the low programming level implied we badly needed higher-level abstractions.

Finally, there was the question of whether programming in a language built on the CTA resulted in fast and portable parallel programs. In principle it should, as argued above. But our line of reasoning had been based on only a single success—imperative languages like C and the RAM model.

⁴ We expected to generalize Fortran to produce Orca-F, and had the idea been successful, we might be discussing Orca-J!

That seemed like the only way to program, and was so familiar that perhaps we'd missed something. Before creating a new language, we needed experience programming with the CTA as the machine model. So, we ran a series of Orca-C experiments examining whether performance and portability could both be achieved.

These were the experiments reported in Calvin Lin's dissertation, *The Portability of Parallel Programs Across MIMD Computers* [22]. He used Orca-C and the CTA to write several parallel programs and measured their performance across five of the popular parallel architectures of the day: three shared memory (Sequent Symmetry, BBN Butterfly GP1000, BBN Butterfly TC2000), and two distributed memory (Intel iPSC-2, nCUBE/7). Though the space of results was complicated, they suggested to us that it was possible to write one generic program based on the behavior of the CTA and produce reasonable performance consistently across vastly different parallel machines.

A further issue concerned the memory model for a new language. The CTA says, in essence, that parallel programming should be attentive to exploiting locality, but the undifferentiated global shared memory that is the logical extension of the von Neumann model's memory ignores locality, and the convenience of ignoring it is the rationale for including hardware support for shared memory. Lin's results said that respecting locality was good for performance and portability, that is, CTA-based programs run well on machines with or without shared memory hardware. But there is the subtle case of shared memory programs written for shared memory hardware computers. It might be that programs developed assuming global shared memory running on shared memory hardware are superior to generic programs focusing on locality on that same hardware. If so, CTA-focused programming might be penalized on such machines. Lin's experiments suggested CTA-based programs were actually better, but not definitively. Ton Ngo [23, 24] addressed the question more deeply and found that for the cases analyzed the CTA was a better guide to efficient parallel programming than global shared memory for shared memory hardware computers. Though it is almost always possible to find an application that uses a hardware feature so heavily that it beats all comers, the opposite question—can the applications of interest exploit the shared memory hardware better using the shared memory abstraction?—was answered to our satisfaction in the negative.

Thus, with the knowledge that performance and portability could be achieved at once and the knowledge that respecting locality produces good results, we began to design ZPL.

7. Initial Design Decisions

Following the celebratory lunch in October 1992, we launched the language design, which allowed us to apply the results of several years' worth of experiments. It was a great relief to me to finally be designing a language. For several years I'd been attending contractors' meetings, workshops and conferences, and listening to my colleagues present their ideas for ways to program parallel computers. Their proposals, though well intended and creative, were

just that: proposals. They were generally unimplemented and unproved. More to the point, they appeared totally oblivious to the issues that were worrying us, making the lack of verifying data even more frustrating.⁵ Now, the preparation was behind us. Our formulation of a high-level language would soon emerge.

7.1 Global View versus Shared Memory

With the results of the Lin and Ngo experiments in mind, we adopted a global view of computation. That is, the programmer would "see" the whole data structures and write code to transform entire arrays rather than programming the local operations and data needed by the processors to implement them. The compiler would handle the implementation. By "seeing" arrays globally and transforming them as whole units, programmers would have power and convenience without relying on difficult-to-implement global shared memory.

To be clear, in shared memory, operands can reference any element or subset of elements of the data space, whereas in the global view operations are applied to entire data aggregates, though perhaps selectively on individual elements using regions.

The problem with shared memory is the "random access," that is, the arbitrary relationship between the processor making the access and the physical location of the memory referenced. The Lin/Ngo results emphasized the importance of locality. When a processor makes an access to a random memory location, odds are that it is off-processor, requiring λ time to complete according to the CTA. Global shared memory gives no structure to memory accesses, and therefore, no way⁶ to maximize the use of the cheaper local memory references.

Some years earlier we had tested the idea of using APL as a parallel programming language. The results of those experiments [25] showed that although most of APL's facilities admitted efficient parallel implementations, the notable exception was indexing. APL permits arrays to be indexed by arrays, $A[V]$, which provides capabilities more powerful than general permutation. Based on usage counts, indexing is heavily used, and although its full power is rarely needed, it would be very difficult to apply special-case implementations to improve performance. We decided at the time that APL's indexing operator was its Achilles' heel. We wanted more control over array reference.

Another indexing technique is "array slices," $A[2:n-1:2]$, as used in, say Fortran 90 [26] and, therefore, in High Performance Fortran [27]. These references are much more structured with their lower bound, upper bound, stride triple. Slices give a simple specification of the data being referenced for a single operand. An effective compiler needs that information, but it also needs to know the relative distribution between the referenced items for (pairs of) operands because it must assign the work on some

⁵ At one point Lin and I tried to engage the community in testing the shared-memory question mentioned above, but it was noticed by only a few researchers [67].

⁶ Many, many proposals have been offered over the years to provide scalable, efficient shared memory, both hardware and software. In 1992 none of these seemed credible.

processor(s), and if the data is not local to the processors doing the work, it must be fetched. So, for example, in the Fortran 90 expression

$$\dots A[1:n] + B[1:n] \dots \quad (1)$$

corresponding elements are added; because of the usual conventions of data placement, this typically would be implemented efficiently without communication. But, the similar expression

$$\dots A[1:n] + B[n:1:-1] \dots \quad (2)$$

sums elements with opposite indices from the sequence. This likely has serious communication implications, probably requiring most of one operand to reference non-local data.

ZPL solved this problem with regions, which give a set of indices for statement evaluation that applies to all array operands; any variation from common specification is explicit. So, in ZPL the computation of expression (1) would be written

$$[1..n] \dots A + B \dots$$

since the operands reference corresponding elements. If the relationship is offset, say by one position, as in $A[1:n-1] + B[2:n]$, then ZPL simply uses

$$[1..n-1] \dots A + B@right \dots$$

where the `@right` effectively adds 1 to each index, assuming the direction `right` has the value `[1]`. This implies (a small amount of) communication, as explained later in the WYSIWYG performance model. With regions, expression (2) requires first that one of the operands be remapped to new indices. Accordingly, the equivalent ZPL is

$$[1..n] \dots A + B#[n-Index1+1] \dots$$

We explicitly reorder using the remap (#) operator and a description of the `n` indices in reverse order (`n-Index1+1`), because the memory model does not permit arbitrary operand references.⁷ Programmers must explicitly move the data to give it new indices.

A mistaken criticism of regions asserts that they are only an alternate way to name array slices, that is, reference array elements. But as this discussion shows, the difference is not embodied in naming, though that is important. Rather, the main difference is in controlling the index relationships among an expression's operands: Regions specify one index set per statement that applies to all operands, modulo the effects of array operators like @; slices specify one index set per operand.

It may appear that this is a small difference since both approaches achieve the same result, and indeed, it may appear that ZPL's global reference is inferior because it requires special attention. But, as argued below in the discussion of the WYSIWYG performance model, ZPL's policy is to make all nonlocal (λ -cost) data references explicit so programmers can know for certain when their data references are fast or slow and what its big-O communication complexity is.

⁷ Indexi is a compiler generated array of indices for dimension i; here it is $1, \dots, n$.

We eliminated general subscripting because we couldn't figure out how to include it and still generate efficient code relative to the performance constraints of the CTA. Because all operands use the same reference base, regions make possible a variety of simplifications that we wanted:

- The compiler could know the relationship among the operands' indices without depending on complex analysis or the firing of a compiler optimization, and could therefore consistently avoid generating unnecessarily general code
- The common case of applying operations to corresponding array elements allows local, 100% communication-free code
- When operands are not corresponding array elements, programmers specify transformations such as the @ or # operators that embody the communication needed to move the elements "into line"
- "Array-wide" index transformations such as remap often admit optimizations like batching, schedule reuse, etc. that use communication more efficiently
- Understanding the performance model and compiler behavior is much more straightforward and, we recognized later, expressing it to programmers is easier (see WYSIWYG model below)

Finally, one happy convenience of regions is that they avoid the repetitive typing of subscripts that occurs with array slices in the common case when subscripts of each operand correspond.⁸

Unquestionably, adopting the global view rather than shared memory drove almost all of the language design. It made the work fun and very interesting. And as we progressed, we liked what we created. Our programs were clear and succinct and errors were rare. It was easy not to miss subscripts.

7.2 Early Language Decisions

Beginning in March 1993 we taught the base language to the Parallel Programming Environments seminar (CSE590-O) students. (The type architecture and CTA had been covered during fall term.) Soon, we worked as a group to extend the language.

As language design proceeded, the question of syntax arose continually. Syntax, with its ability to elicit heated (verging on violent) debate carries an additional complication in the context of parallel languages. When syntax is familiar, which usually means that it's recognizable from some sequential language, it should have the familiar meaning. A `for`-loop should work as a `for`-loop usually works. Accordingly, parallel constructs will require new or

⁸ Once, to make the point that repeating identical or similar index references using slices is potentially error prone, we found a 3D Red/Black SOR program in Fortran-90, filled with operands of the form $A[lo_x+1:hi_x+1:2, lo_y-1:hi_y-1:2, lo_z-1:hi_z-1:2]$, and then purposely messed up one index expression. Our plan was to ask readers which operand is wrong. It was nearly impossible to recognize the correct from the incorrect code, proving the point but making the example almost useless once we had forgotten which program was messed up. Which was which? In frustration, I eventually dumped the example.

distinctive syntax, motivating such structures as **parabegin/paraend**. The new semantics have a syntax all their own, but this further raises the barrier to learning the new language. In our design discussions we tended to use a mix of simplified Pascal/Modula-based syntax with C operators and types. It was an advantage, we thought, that Pascal was by then passé, and was thus less likely to be confused with familiar programming syntax. With only a little discussion, we adopted that syntax.

Perhaps the two most frequently remarked upon features of ZPL’s syntax are the use of `:=` for assignment and the heavy use of **begin/end** rather than curly braces. Of course, the use of `:=` rather than `=` was intended to respect the difference between assignment and equality, but it was a mistake. ZPL’s constituency is the scientific programming community. For those programmers such distinctions are unimportant and `:=` is unfamiliar. Programmers switching between ZPL and another language—including the implementers—regularly forget and use the wrong symbol in both cases. It’s an unnecessary annoyance.

The grouping keywords **begin/end** were adopted with the thought that they were more descriptive and curly braces would be used for some other syntactic purpose in the language. Over the years of development curly braces were used for a number of purposes and then subsequently removed on further thought. In the current definition of the language, curly braces are used only for the unimportant purpose of array initialization. In retrospect, their best use probably would have been for compound statements.

7.3 Early Compiler Decisions

Language design continued during the summer of 1993 in parallel with the initial compiler implementation. To solve the instruction set portability problem we decided to compile ZPL to C, which could then be compiled using the parallel platform’s native C compiler to create the object code for the processor elements. Making the interprocessor communication portable requires calls to a library of (potentially) machine-specific transport routines. Though this problem would ultimately be solved by the Ironman Interface [28], it was handled in 1993 by emitting *ad hoc* message passing calls.

To get started quickly, we began with a copy of the Parafrase compiler from the University of Illinois [29]. This gave us instant access to a symbol table, AST and all the support routines required to manipulate them. With a small language and borrowed code, the compiler came up quickly. The compiler’s runtime model followed the dictates of the CTA, of course, and benefited greatly from the Orca-C experience.

One measure of success was to demonstrate performance and portability from our high-level language. Though there is a natural tendency in software development to “get something running” and worry about improving performance (or portability) later, we took as a principle that we would deliver performance and portability from the start, Policy #2. We reasoned that this would ensure that the language and compiler would always have those properties: Preserving them would be a continual test on the new features added to the language; any proposed feature harming performance or limiting portability wouldn’t be included.

In September, 1993, the basic set of features was reported in “ZPL: An Array Sublanguage” at the Workshop on Languages and Compilers for Parallel Computers

(LCPC) [5]. The idea of building an array sublanguage—contained, say, in Orca-C or other general parallel facilities—was crucial initially because it meant that we could focus on the part of parallel computation we understood and could do well, such as the inner loops of Simple and other benchmarks. *We didn’t have to solve all the problems at once.* Because we had dumped shared memory and adopted the demanding CTA requirements, there was plenty to do. Over time, the language got more general, but by late fall of 1993—a year after Lin’s dissertation defense—we were generating code, and by the next summer, when 1994 LCPC came around, we were demonstrating both performance and portability on small parallel benchmarks [6].

7.4 Other Languages

As noted above, we had already determined some years earlier [25] that APL was an unlikely candidate as a parallel programming language. Nevertheless, because it is such an elegant array language we often considered how our sample computations might be expressed in APL. On hearing that ZPL is an array language, people often asked, “Like APL?” “No,” we would say, “ZPL is at the other end of the alphabet.” The quip was a short way to emphasize that parallelism had caused us to do most things quite differently. (A related quip was to describe ZPL as the “last word in parallel programming.”)

C*, however, was a contemporary array language targeted to parallel computers, and therefore had to deal with similar issues [30]. A key difference concerned the view of a “processor.” Being originally designed for the Connection Machine family, C* took the virtual processor view: that is, in considering the parallel execution of a computation programmers should imagine a “processor per point.” The physical machine may not have so much parallelism, but by multiplexing, the available parallelism would be applied to implement the idealization. The idea is silent on locality. The CTA implies that a different processor abstraction would be preferable, namely, one in which a set of related data values all reside on the same processor. The CTA emphasizes the benefits of locality since nonlocal references are expensive.

Of course, the processor-per-point (1ppp) view and the multiple points per processor (mppp) view are identical when the number of data values and the number of processors are the same ($n=P$), which never happens in practice. Still, because implementations of 1ppp models bundle m values on each processor for $n = mP$ and multiplex the 1ppp logic on each value, it has been claimed the 1ppp models are equivalent to ZPL’s mppp model. They are not. As one contradicting example, consider the array A declared over the region $[1 \dots n]$ and consider the statement

```
[1..n-1] A := A@east;
```

which shifts the values of A one index position to the left, except for the last. The 1ppp model abstracts this operation as a transmission of every value from one virtual processor to its neighbor, while ZPL’s mppp model abstracts it as a transmission of a few “edge elements” followed by a local copy of the data values. (See the WYSIWYG model discussion below for more detail.) All communications in both models conceptually take place simultaneously and are therefore equivalent. But the data copy is not captured by the 1ppp model, though it is probably required by the im-

plementation; the data copy is captured by the mppp model. This repositioning of the data may seem like a small detail, especially considering our emphasis on communication, but it can affect the choice of preferred solutions. In the Cannon v. SUMMA matrix multiplication comparison [31] described below, repositioning has a significant effect on the analysis. This distinction between the virtual processor and the CTA's multiple points per processor kept us from using C* as a significant source of inspiration.

The High Performance Fortran (HPF) effort was contemporaneous with ZPL, but it had no technical influence. HPF was not a language design,⁹ but a sequential-to-parallel compilation project for Fortran 90. Because we were trying to express parallelism and compile for it, that is parallel-to-parallel compilation, we shared few technical problems. And we were building on the CTA and they were not: On the first page of the initial HPF design document [27] the HPF Forum expressed its unwillingness to pick a machine model—"Goal (2) Top performance on MIMD and SIMD computers with non-uniform memory access costs (while not impeding performance on other machines)." At the time I assumed that the Goal had more to do with the need to design by consensus than any neglect of the literature. Either way, attempting to parallelize a sequential language and doing so without targeting the CTA were blunders, preventing us from having much interest in its technical details. But, with most of the research community apparently signed on, we came to see ourselves as competitors because of the divergent approaches. We compared performance whenever that made sense. We were proud to do better nearly always, especially considering the huge resource disparities.

Finally, ZPL is apparently a great name for a language. In the summer of 1995 we received a letter from attorneys for Zebra Technologies Inc. alleging trademark infringement for our use of "ZPL" for a programming language, and threatening legal action. Zebra is a Midwest-based company making devices for UPC barcoding including printers, and they had trademarked their printer control language, Zebra Programming Language, ZPL. We had never heard of them, and we guessed they had never heard of us until MetaCrawler made Web searches possible and they found us via our Web page. Our response pointed out that theirs is a control language for printers and ours was a supercomputer language, and that the potential audiences do not overlap. Nothing ever came of the matter, but every once in a while we get a panic question from some harried barcode printer programmer.

8. Creating ZPL's Language Abstractions

In this section we recap the high points of the initial design discussions and the important decisions that emerged during the first 18 months of project design.

⁹ The statement may be unexpected, but consider: Beginning with Fortran 90 and preserving its semantics, HPF focused on optional performance directives. Thus, all existing F-90 programs compute the same result using either an F-90 or HPF compiler, and all new programming in HPF had the same property, implying no change in language, and so no language design.

8.1 Generalizing Regions

Regions as a means of controlling array computation were a new idea, and working out their semantics percolated through the early design discussions. It was clear that scoping was the right mechanism for binding the region to the arrays. That is, the region applying to a d -dimensional array is the d -dimensional region prefixing the statement or the closest enclosing statement. So in

```
[RegOuter] begin
    Statement 1;
[RegInner]   Statement 2;
    Statement 3;
end;
```

d -dimensional arrays in *Statement 2* are controlled by region *[RegInner]*, while d -dimensional arrays in statements 1 and 3 are controlled by region *[RegOuter]*. This region is known as the **applicable region** for the d -dimensional arrays of the statement. Regions had been strongly motivated by scientific programming style, where it is common to define a problem space of a fixed rank, e.g. 2D fluid flow, and perform all operations on that state space. With large blocks of computation applying to arrays with the same dimensionality, scoped regions were ideal.

Because regions define a set of indices of a specific dimensionality, arrays of different dimensionalities require different regions. ZPL's semantics state that the region defines the elements to be updated in the statement's assignment, so initially it seemed that only one region was needed per statement. (Flooding and partial reduction soon adjusted this view, as explained below.) But, because statements within a scope may require different regions, the original language permitted regions of different ranks to prefix a single statement, usually a grouping statement, as in

```
[1..n] [1..n, 1..m] begin ... end;
```

Though this feature seemed like an obvious necessity, it was rarely if ever used during ZPL's first decade and was eventually removed.

Because the team's *modus operandi* was to use a few specific example computations to explore language facilities and compiler issues, we showed the language to campus users from an early stage as a means of finding useful examples. This engendered a surprising (to me) modification to the language. Specifically, we had defined a region to be a list of index ranges enclosed in brackets, as in the declaration

```
region R = [1..n, 1..n];
```

and required that to control the execution of a statement the name appear in brackets at the start of the line, as in

```
[R] A := ... ;
```

It seemed obvious to us as computer scientists that when a region was specified literally on a statement that it should have double brackets, as in

```
[[1..n, 1..n]] A := ... ;
```

Users were mystified why this should be and were not persuaded by the explanation. We removed the extra brackets. Curiously, computer scientists have since wondered why ZPL doesn't require double brackets for literal regions!

8.2 Directions

The original use of directions as illustrated in Figure 2 was to translate a region to refer to nearby neighbors, as in $A@\text{west}$. It is a natural mechanism to refer to a whole array of adjacencies, i.e. the array of western neighbors, and contrasts with single-point alternatives such as $A[., .-1]$. Generally, directions were a straightforward concept at the semantic level.

The compiler level was another matter. Directions refer to elements some of which will necessarily be off-processor. (A complete description of the runtime model is given in the WYSIWYG section below.) This presents several problems for a compiler. First, communication with the processor storing the data is required, and the compiler must generate code to access that data, as illustrated in Figure 4. Obviously, the direction is used to determine which is the relevant processor to communicate with. Second, in order to maximize the amount of uninterrupted computation performed locally, buffer space—called **fluff** in ZPL and known elsewhere as shadow buffers, ghost regions or halo regions—was included *in position* in the array allocation. Again, it is obvious that the direction indicates where and how much fluff is required. So, for example, with $\text{north} = [-1, 0]$, the (direction) vector has length 1 and so only a single row can be referenced on a neighboring processor. When the direction is $\text{west2} = [-2, 0]$ the compiler introduces a double row of fluff.

Unlike regions, however, directions were not originally allowed to be expressed literally, as in $A@[0, -1]$. Furthermore, not allowing literal directions prevented the temptation to allow variables in a direction, as in $A@[x, y]$. The problem with $A@[x, y]$ is that we don't know how much fluff to allocate, and we don't know how much communication code to generate. For example, $A@[1, 1]$ requires fluff on the east, southeast and south edges of the array and three communications (generally) to fill them, while $A@[0, 1]$ requires only an east column of fluff and one communication. Effectively, ZPL's @-translations were only statically specified constant translations. By knowing the off-processor references exactly, fluff could be allocated at initialization, avoiding array resizing, and the exact communication calls could be generated. Though these restrictions were (not unreasonably) criticized by programming language specialists, they were only rarely a limitation on using the language. One such example was the user whose fast multipole method stencil required 216 neighbor references.

As indicated in Figure 4, the @ operator will reference whole blocks of data, e.g. columns, and these can be transmitted as a unit. Batching data values is a serious win in parallel communication, because of the overhead to set up the transfer and the advantages of the pipelining of the transmission itself. ZPL gets this batching for free.¹⁰ But we could do even better knowing at compile time what communication is necessary. For example, we move communication operations earlier in the control flow to pre-

fetch data; we also determine when data for several variables is moving between the same processors, allowing the two transmissions to be bundled, as described below. These scheduling adjustments to the default communication lead to significant performance improvements [32] (see Ironman, below). In essence, the language enforced restrictions on @ to give a fast common case.

Having been conservative with directions originally allowed us to work out fully the compilation issues. Confident that we knew what we were doing, we added literal directions.

8.3 Region Operators

A key question is whether a procedure must specify its own region, or whether it can inherit its region from the calling context. That is, are regions supplied to procedures statically or dynamically? Inheriting was chosen because it is more general and provides a significant practical advantage to programmers in terms of reuse. Since portions of the inherited region will be used in parts of the computation, it is obviously necessary to define regions from other regions.

Specifically, let region $R = [1..n, 1..n]$ and direction $\text{west} = [0, -1]$. Then to refer to boundary values of the region beyond R's west edge, i.e. a 0th column, is specified $[\text{west of } R]$ (see Figure 5). The *region operators*—informally known as the *prepositional operators* because they include *of*, *in*, *at*, *by* and *with*—

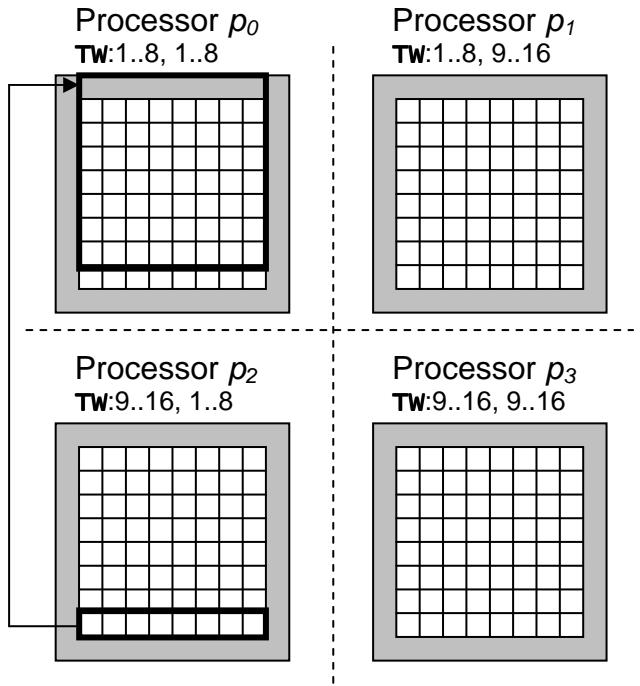


Figure 4. Schematic of the block allocation of array TW from Figure 2, where $n=16$ and the $P=4$ processors have a 2×2 arrangement. The array is shown as a grid. The fluff buffers, shown shaded, completely surround the local portion of the array because there are @-references in all eight directions. In the diagram, local portion of the (operand) array $TW@\wedge no$ for processor p_0 is shown in heavy black; because the north neighbors of the top row wrap to reference the bottom row on processor p_2 , communication from p_2 to p_0 is required to fill the fluff buffer.

¹⁰ There was a flutter of compiler research at one point focused on techniques to determine when coalescing communications is possible [68]; raising the semantic level of the language eliminates the problem.

generally take a direction and a region of the same rank and produce a new region, which can be used with other region operators for region expressions, as in `[east in north in R]` for the upper right corner element of a 2D region R . The operator `at` translates regions just as `@` translates indices for operands; `by` strides indices, as in `[1..n by 2]`, which selects odd indices in this example. Such operators allow regions to be defined in terms of other regions without the need to know the actual limits of the index range. Similarly, we included the *ditto* region, as in `[]`, meaning to inherit the dynamically enclosing region, and *empty dimensions*, as in `[1..n,]`, meaning to inherit the index range of that dimension from the enclosing region. Defining the prepositional operators and considering regions in general led to a 4-tuple descriptor for index ranges that probably has applicability beyond ZPL [33].

The `with` operator, as in `[R with mask]`, uses a Boolean array expression (`mask`) rather than a direction with a region as the other prepositional operators do. The variable is a Boolean array of the same rank as R specifying which indices to include, and making it similar to the `where` of languages like Fortran 90. Though there are ample cases where masking is useful and sensible, the ZPL team, especially Brad Chamberlain, was always bothered by the fact that the execution of a masked region would be proportional to the size of R rather than the number of the true values in the mask. This ultimately motivated him to develop a general sparse array facility; `by` was also a motivation to develop multiregions (see below).

8.4 Reduce and Flood

Since APL, reduce—the application of an associative operator to combine the elements of an array—has been standard in array programming languages. (Scan is also

common and is discussed below.) The key property from our language design standpoint is the fact that reduce takes an array as an operand but produces a scalar as a result. This is significant because ZPL has been designed to combine like-ranked arrays to produce another array of that rank. (The reasons concern the runtime representation and the WYSIWYG model, as explained below.) The original ZPL design only included a “full” reduction ($op \ll <$) that combined all elements into a scalar, where op is $+$, $*$, $\&$, $|$, \min , \max . We had plans to include a partial reduction, but had not gotten to that part of the design. Thus, initially, the fact that reduce reduces the rank of its operand was simply treated as an exception to the like-ranked-arrays rule. (Scalars were exceptional anyway, because they are replicated on all processors whereas arrays are distributed.) When it came time to add partial reduction, the issue of an “input shape” and an “output shape” had to be addressed.

Partial Reduce A **partial reduction** applies an associative operator ($+$, $*$, $\&$, $|$, \max , \min) to reduce a *subset* of its operand’s dimensions. The original ZPL solution, motivated by APL’s $+/[2]A$, was to specify the reducing dimension(s) by number. This is a satisfactory solution for APL because the **reduced dimensions** are removed. In ZPL they remain and are assigned a specific index to produce a **collapsed dimension**; a dimension is collapsed if the index ranges over a single value, as in `[1..n, 1..1]`, which is accepted shorthand for `[1..n, 1..1]`.

The consequence of these semantics is that a region is needed to specify the indices of the operand array and another region is needed to specify the indices of the result array. Thus, we write

`[1..m, 1] B := +<< [1..m, 1..n] A;`

to add the elements of A ’s rows and store the results in B ’s

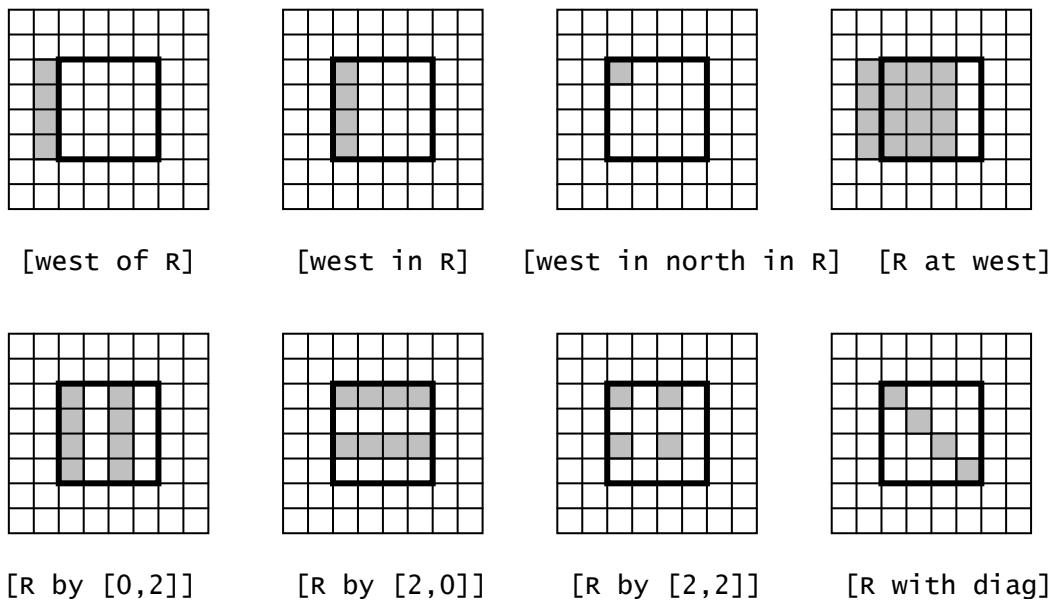


Figure 5. Schematic diagram of various region operators. Referenced elements are shaded; R is a region, $\text{west} = [0, -1]$, $\text{north} = [-1, 0]$, diag is a Boolean array with trues on the diagonal.

first column. The compiler knows this by comparing the two regions, noting that the first dimension is unchanged and the second is collapsed and so must be combined; the reduction operator ($+<<$) provides the addition specification.

The reason to depart from standard practice by keeping the reduced dimensions as collapsed dimensions rather than removing them and thereby lowering the dimensionality of the result array relative to the operand array was to manage allocation and communication efficiently. When, for example, a 2D array collapses to a 1D array by a partial reduction, how do we align the result relative to the operand array? Both the compiler writer and the user need to know. Normally, the result will interact in subsequent computations with other arrays having the operand shape. So aligning our 1D result with 2D arrays is rational. But how?

Aligning the result with columns is possible; aligning it with rows is possible; treating it as a 1D array is also possible, though because ZPL uses different allocations for 1D and 2D arrays (in its default allocations), the operand and result arrays would misalign. The decision matters because in addition to the time required to compute the partial reduction, each choice entails different amounts of communication to move it to its allocated position, ranging from none, to an additional transpose, to the very expensive general remap.

By adopting collapsed dimensions, however, the result aligns with the operand arrays in the most efficient and natural way: it aligns with the “remaining” shape; further, the user specifies (by the index(es)) exactly how. Most importantly, collapsed dimensions allow us to bound the communications requirements to $O(\log P_i)$ where P_i is the number of processors spanning the reduced dimension i . Because of such analysis, the powerful partial reduction operator is also an efficient operator. (This type of analysis—worry about communication, worry about how the user controls the computation—characterized most of our design discussions about most language features.)

This “two region” solution is elegant, but we didn’t figure it out immediately.

Creating Flood Indeed, we came to understand partial reduction by trying to implement its inverse, flood. **Flood** replicates elements to fill array dimensions. Flood is essential to producing efficient computations when arrays of different rank are combined, e.g. when all columns of an array (2D) are multiplied by some other column (1D). In the case of flood the need for two regions is probably more obvious because there must be a region to specify the source data and another region to specify the target range of indices for the result. Thus, to fill **A** with copies of **B**'s first column, write

```
[1..m, 1..n] A := >> [1..m, 1] B;
```

The duality with partial reduction is obvious.

The use of two regions—one on the statement to control the assignment and the overall behavior of the computation, and one on the source operand specifying its indices—was a difficult concept to develop. Flooding was a new operation that I more or less sprung on the team at a ZPL retreat in 1994; it was introduced as a new operator and engendered spirited debate. The duality with partial reduction was not immediately recognized because partial reduction still had its “APL form.” Flooding was not adopted at the retreat, and Brad Chamberlain and I contin-

ued to kick it around for a couple of weeks more after the rest of the team had tired of the debate. In our conversations the solution emerged, the duality emerged, the elimination of partial reduction’s dimensions-in-brackets definition was adopted, and the idea of recognizing which dimensions participate in the operations by comparing the two regions and finding which dimensions are collapsed/expanded was invented.

This experience with flood represented in my view the first time we had attacked a problem “caused” by trying to build a global view language for the CTA, and had produced a creative solution that advanced the state of the art in unexpected ways. Regions, directions, etc. were new ideas, too, but they “fell out” of our approach. Flooding posed significant language and compiler challenges, and we’d solved them with creativity and hard work. Flooding, which motivated a new algorithmic technique (problem space promotion [34], discussed below), continues to be a source of satisfaction.

Flood Dimensions Shortly after inventing the two region solution, we invented the concept of the flood dimension. To appreciate its importance, consider why parallel programmers are interested in the flooding operation in the first place. Imagine wanting to scale the values in the rows of a 2D array to the range [-1,1]. This is accomplished if for each row the largest magnitude is found and all elements of the row divided by that value. ZPL programmers write

```
[1..m,1..n] A := A /  
&gt;> [1..m,1] max<< [1..m,1..n] abs(A));
```

which says, use as the denominator the flooded values of the maximum reduction of each row. That is, find the maxima, replicate those values across their rows, and perform an element-wise division of the replicated values into the row. The statement allows the ZPL compiler

- to treat the row maxima as a column and transmit it to the processors as a unit, achieving the benefits of batched communication rather than repeated transmission of single values,
- to multicast rather than redundantly transmit point-to-point, and
- to cache, since each processor will store its section of the column and repeatedly use the values for the multiple references local to it.

In effect, the programmer’s use of flood described the computation in a way that admitted extremely efficient code.

The odd feature of the last statement is the use of “1” in the flood,

```
&gt;> [1..m,1] max<< [1..m,1..n] abs(A))  
^
```

specifying that the result of the max reduction is to be aligned with column 1 before flooding. Why column 1? In fact there is no reason to assign the result of the reduction

to any specific column.¹¹ What we want to say is that a dimension is being used for flooded data, and so no specific index position is needed—or wanted. This is the flood dimension concept, and is expressed by an asterisk in the dimension position, as in `[1..m, *]`. In such regions any column index conforms with the flood dimension.

The flood dimension extends regions with an intellectually clean abstraction, but there is a greater advantage than that. The compiler can implement a flood dimension extremely efficiently by allocating a single copy of that portion of the replicated data needed by a processor. So, for example, if `CellCountPerRow` is declared as the flood array `[1..n, *]` for the allocation shown in Figure 4, the assignment

```
[1..n, *] CellCountPerRow := +<< [1..n, 1..n] TW;
```

adds up the number of cells in each row and stores the relevant part of `CellCountPerRow` on each processor. That is, referring to Figure 4, both processors p_0 and p_1 would store only the first eight elements of `CellCountPerRow`, and p_2 and p_3 would store the last eight. These “logically populated” arrays are an extremely efficient type of storage, and do not require a reduction or flood to assign them. The next section gives an elegant example.

8.5 Flooding the SUMMA Algorithm

The power and beauty of flooding became clear in 1995 when we applied it to matrix multiplication. We had written several row-times-column matrix product programs in ZPL to test out language features, when Lin reported a conversation with Robert van de Geijn of UT Austin about his new algorithm with Jerrel Watts, which they called SUMMA—scalable, universal matrix multiplication algorithm [35]. Their idea was to switch from computing dot-products as a basic unit of a parallel algorithm to computing successive terms of all dot-products at once. They claimed that it was the best machine independent matrix multiplication algorithm known.

To see the key idea of SUMMA and why flooding is so fundamental to it, recall that in the computation $C = AB$ for 3×3 matrices A and B , the result is

$$\begin{aligned} C_{1,1} &= A_{1,1}B_{1,1} + A_{1,2}B_{2,1} + A_{1,3}B_{3,1} & C_{1,2} &= A_{1,1}B_{1,2} + A_{1,2}B_{2,2} + A_{1,3}B_{3,2} \\ C_{1,3} &= A_{1,1}B_{1,3} + A_{1,2}B_{2,3} + A_{1,3}B_{3,3} \\ C_{2,1} &= A_{2,1}B_{1,1} + A_{2,2}B_{2,1} + A_{2,3}B_{3,1} & C_{2,2} &= A_{2,1}B_{1,2} + A_{2,2}B_{2,2} + A_{2,3}B_{3,2} \\ C_{2,3} &= A_{2,1}B_{1,3} + A_{2,2}B_{2,3} + A_{2,3}B_{3,3} \\ C_{3,1} &= A_{3,1}B_{1,1} + A_{3,2}B_{2,1} + A_{3,3}B_{3,1} & C_{3,2} &= A_{3,1}B_{1,2} + A_{3,2}B_{2,2} + A_{3,3}B_{3,2} \\ C_{3,2} &= A_{3,1}B_{1,3} + A_{3,2}B_{2,3} + A_{3,3}B_{3,3} \end{aligned}$$

Notice that the first term of all of these equations can be computed by replicating the first *column* of A across a 3×3 array, and replicating the first *row* of B down a 3×3 array, that is, flooding A 's first column and B 's first row, and then multiplying corresponding elements; the second term results from replicating A 's second column and B 's second row and multiplying, and similarly for the third term.

The ZPL program for SUMMA (Figure 6) applies flooding to compute these terms. It begins with declara-

```
var A : [1..m,1..n] double; Left operand
B : [1..n,1..p] double; Right operand
C : [1..m,1..p] double; Result array
Col : [1..m,*] double; Col flood array
Row : [*..1..p] double; Row flood array
...
[1..m,1..p] C := 0.0; Initialize C
for k := 1 to n do
[1..m,*] Col := >>[ ,k] A; Flood kth col of A
[*..1..p] Row := >>[k, ] B; Flood kth row of B
[1..m,1..p] C += Col*Row; Multiply & accumulate
end;
```

Figure 6. ZPL matrix multiplication program using the SUMMA algorithm.

tions, including `Col` and `Row`, flood dimension arrays. These two arrays can be assigned a column and a row, respectively, which is replicated in the `*` dimension. After initializing `C`, a `for`-loop steps through the columns and rows of the common dimension, flooding each and then accumulating their element-wise product, implementing SUMMA.

We were delighted by the van de Geijn and Watts algorithm, because it was not only the best machine-independent parallel matrix product algorithm according to their data, it was the shortest matrix multiplication program in ZPL. (The three statements in the `for`-loop can be combined,

```
[1..m,1..p] for k := 1 to n do
    C += (>>[ ,k] A )*( >>[k, ] B );
    end;
```

eliminating the need for the `Col` and `Row` arrays.) When the best algorithm is also the easiest to write, the language's abstractions match the problem domain well.¹²

8.6 Epic Battle: Left-hand Side @s

At the 1994 ZPL retreat mentioned above, the issues surrounding flood were not resolved, first because they were deep and complex, requiring thought and reflection, and second because we spent most of our time arguing over left-hand side @s. All (early) project members agree that this was our most contentious subject, and the magnitude of the disagreement has now grown to mythic proportions. Here's what was at issue.

ZPL uses @-references to specify operands in expressions, as in

```
[2..n-1,2..n-1] B := (A@north + A@east
+ A@west + A@south)/4;
```

where directions translate the region's indices to a new region to reference the operand. Programming language design principles encourage the consistent use of constructs, and so because @-translations are allowed on the right-hand side of an assignment, they should be allowed

¹¹ ZPL recognizes the reduction-followed-by-flood as an idiom and permits `>> op <<` with no dimension specified, but that doesn't solve the general problem.

¹² The original SUMMA is blocked. ZPL's is not, and blocking complicates the ZPL significantly.

on the left-hand side as well. The semantics of left-hand side @ are intuitive and are natural for programmers, as in

```
[2..n,2..n] B@nw := A@north & A & A@west;
```

which flows information into the corner of an array when used iteratively, for example, in a connected components algorithm [36]. Of course, left-hand side @s are not required, since the programmer could have written

```
[1..n-1,1..n-1] B := A@south & A@se @ A@east;
```

But he or she may think of the computation from the other point of view. Another principle of language design says to avoid placing barriers in the programmer's way.

The compiler writer's viewpoint noted that ZPL's semantics use the region to specify which indices are updated in the assignment. Processors are responsible for updating the values of the region that they store. By using a left-hand side @ the indices that a processor is to update are offset. Which processor is responsible for updating which elements—only those it owns or those it has the data for? Should the value for a location be computed on another processor and then shipped to the processor that stores it; or should every processor simply handle the values it stores? The presence of such a construct in the language complicates the compiler and has serious optimization implications.

Like many issues that engender epic design battles, left-hand side @s are an extremely small matter. How and why they excited so much controversy is unclear. It may simply be that the particular circumstances of the situation provided the accelerant to heat up the discussion, and that on another day the issue would have hardly raised comment.¹³ Whatever the reason, left-hand side @s were ZPL's biggest controversy. In this case, however, everyone won: The language includes left-hand side @s and the compiler at that time automatically rewrote any use of left-hand side @s to remove them, leaving the remainder of the compiler unaffected. In the years since a direct implementation has been developed.

8.7 A Bad Idea From Good Intentions

As mentioned above, the Simple benchmark written in Orca-C devoted more than half its lines to computing boundary values [21]. This represents a huge amount of work for programmers. A high-level language should help. ZPL's regions, directions and wrap-@ constructs do help significantly, and almost eliminate concern about boundaries. But we added one more feature, automatic allocation of boundary memory. Automatic allocation was a mistake.

Recall that the **of** prepositional operator refers to a region beyond the base region, that is,

```
[west of R] A := 0.0; -- Initialize west boundary
```

Programmers were confused by this unfamiliar idea, not knowing when it did or didn't apply to an **of**-region. Defining when auto-allocation made sense was complicated

¹³ Perhaps not. An early project member reviewing this manuscript complained that my characterization of it as a “small matter” revealed my biases, and then went on to emphasize its depth by reiterating all of the “other side’s” arguments ... a dozen years later!

because it applied only to **of**-regions extending regions used in array allocation, though **of**-regions are used in many other circumstances. This produced a hard-to-follow set of definitional conditions. Even when users understood them, auto-allocation seemed not to be used. Basically, it violates what should be a fundamental principle of language design: *Make the operations clear and direct with no subtle side effects*. Automatic allocation was a side effect to an array expression. It was deprecated and eventually removed.

8.8 Working With Users

Throughout the ZPL development we had the pleasure of working with a group of very patient and thoughtful users, mostly from UW. Often their comments and insights were amazing, and they definitely had impact on the final form of the language.

Adding Features Direct complaints about the language were the least likely input from users, though that may have reflected their discretion. The most direct comments concerned facilities that ZPL didn't have. “ZPL needs a complex data type,” was a common sentiment among users and, like good computer scientists, we replied that users could implement their own complex arithmetic using existing facilities. But such responses revealed the difference between tool builders and tool users, and being the builders, we added **complex**. Quad-precision was another example. Also, Donna Calhoun, along with others from UW Applied Math, strongly encouraged our plans to add sparse arrays; when Brad Chamberlain implemented them, she was the first user.

Shattered control flow is an example of a language construct that emerged by working with a user, George Turkiyyah of UW Civil Engineering. Shattered control flow refers to the semantics of using an array operand in a control flow expression. Normally, when a control statement predicate is a scalar, say **n**, ZPL executes one statement at a time, as in

```
if n < 10000 then baseCase(n, A);
```

but if the control predicate evaluates to an array of values, as it would if **A** is an array in

```
[R] if A < 0 then A := -A;
```

the control flow is said to shatter into a set of independent threads evaluating the statement(s) in parallel for each index in **R**. In the example, the conditional is applied separately for each index in the region; for those elements less than 0, the **then**-clause is executed; for the others it is skipped. Turkiyyah and I were having coffee, discussing how his student might solve a problem in ZPL and making notes on a napkin. In the discussion I had used several such constructs without initially realizing that they didn't actually match current ZPL semantics, which didn't allow non-scalar control expressions. But they were what was required for his problem, and knowing a solution [30, 37], I'd used it. When I thought about it later, it was clear that this was an elegant way to include a weak **parabegin/paraend** construct in ZPL's semantics. By the time the student's program was written, shattered control flow was in the language.

The Power of Abstraction The best user experiences were of the “satisfied customer” type, and two stand out as more notable for their elegance. The first was a graduate

student in biostatistics, Greg Warnes, who came by one day with a statistics book containing a figure with a half dozen equations describing his computation. He said that his sequential C program required 77 lines to implement the equations, while his ZPL program required only 11 lines. Moreover, he emphasized, the ZPL was effectively a direct translation from the math syntax of the book to the syntax of ZPL. He may have been even more delighted with his observation that day than we were.

In the second case, I'd given a lecture about ZPL to an audience in Marine Sciences on Thursday, and by the next Monday one of the faculty emailed a ZPL program for a shallow water model of tidal flow. Reading over the several pages of the program, I was astonished to notice that he'd used several named regions, all containing the same set of indices. Most programs use multiple regions, of course, but one name per index set is sufficient. What the shallow-water program revealed was the programmer's use of regions as abstractions in the computation—different region names referred to different phenomena. The appropriate assignment of indices made them work, and the fact that they might name the same set of indices was irrelevant.

Tool Developers It was a surprise to realize that a programming language is an input to other projects. At the University of Massachusetts, Lowell, ZPL was revised for use in a signal processing system. At the University of Oregon, ZPL was instrumented by Jan Cuny's group to study program analysis and debugging. At Los Alamos National Laboratory, ZPL was customized to support automatic check-pointing, an important concern for long running programs and a challenge for decentralized parallel machines.

David Abramson and Greg Watson of Monash University in Melbourne, Australia applied the interesting concept of relative debugging to ZPL programs [38]. **Relative debugging** uses a working program to specify correct behavior for finding errors in a buggy program. Relative debugging is a powerful tool in parallel programming because parallel programs often implement the computation of a working sequential program. Errors in the parallel program execution can be tracked down *relative* to the sequential program's execution by running both simultaneously and asserting where and how their behaviors should match. The tool is very convenient.

The demonstrations of relative debugging were impressive. Watson found a bug in the ZPL version of Simple, one in the sequential C version and one in both! The error in ZPL was an extra delta term in computing the heat equation; this problem had been in every parallel version of Simple we had ever written. (The only effect fixing the bug would have had on published timing results would have been to improve them trivially.) The bug in the C program was to leave out the last column in one array computation. Watson said, “Whoever wrote the ZPL code obviously noticed this and corrected for it (or the ZPL syntax corrected it automatically).” But such “off-by-1” errors are hard to make once the named regions are set up correctly. Finally, the programs disagreed on how they computed the error term controlling convergence, and in this regard they were both wrong! Referring to the original Simple paper [39] allowed Watson to fix both.

9. WYSIWYG Performance Model

Though the CTA was always the underlying “target” computer for ZPL, the idea of explicitly handing programmers a performance model was not originally envisioned. In retrospect, it may be ZPL’s best contribution. Notice that as the compiler writers, we had had the performance model in our heads the whole time. The achievement was that we recognized that it could be conveyed to programmers.

9.1 The Idea

How the idea emerged has been lost, but I believe it was an outgrowth of a seminar, *Scientific Computation in ZPL*, which we ran every winter quarter beginning in 1996. These seminars were a chance for us to present parallel programming to scientists and engineers “properly,” i.e. in ZPL. The format was for project members to teach ZPL and for students to write a program of personal interest to be presented at the end of the term. The students, unlike researchers and seasoned users who grabbed our software off the Web, needed to be told how to assess the quality of a ZPL program. Conveying the execution environment motivated our interest in a model.

The champion of WYSIWYG was E Chris Lewis, who had joined the project a year after the initial team and saw ZPL with fresh eyes. He spoke of a concept “manifest cost” for an operation, which seemed pretty obvious to the implementers. But he persisted and as we worked the idea over, it became clear how valuable specifying costs could be for users. The model was documented in Chapter 8 of *A Programmer’s Guide to ZPL* [40], which was the main language documentation at the time and was eventually published in 1999.

The **What You See Is What You Get (WYSIWYG)** performance model is a CTA-based description of the performance behavior of ZPL language constructs expressed with syntactically visible cues [31]. The WYSIWYG model specifies what the parallelism will be and how the scalar computation and communication will be performed, telling programmers how ZPL runs on the CTA and by extension on the machines the CTA models. This performance specification cannot be given in terms of time or instruction counts because the programs are portable and every platform is different; rather, it gives relative performance.

To illustrate how it works, consider the two assignment statements from the Life program (Figure 2):

```
NN := TW@^nw + TW@^no + TW@^ne
      + TW@^w           + TW@^e
      + TW@^sw + TW@^so + TW@^se;
```

```
TW := (TW & NN = 2) | (NN = 3);
```

How do they perform? They look similar, and the semantics say that element-wise arithmetic operations on arrays are performed atomically in a single step. But, we know that “step” will be implemented using many scalar operations on a CTA. How will this be done?

Programmers know from the WYSIWYG model that although both statements will be performed fully in parallel, and that the n^2 scalar operations implementing each of them are only negligibly different in time cost (seven binary ops versus four), the two statements actually have noticeably different performance. This is because the assignment to

NN requires point-to-point communication, thereby incurring at least λ additional cost, while the assignment to TW uses no communication at all.

How is this known? *Every use of @ entails point-to-point communication according to the WYSIWYG model, and the absence of @ (and other designated array operators) guarantees the absence of communication.* By looking for these cues, programmers know the characteristics of the statements: The first statement requires communication, the second does not. Further, the amount of communication—total number of values transferred and where—is also described to the programmer based on other program characteristics. Knowing when and how much communication a statement requires is crucial, because although it cannot be avoided, alternative solutions can use vastly different amounts of it. ZPL programmers can analyze performance as they write their code, judge the alternatives accurately and pick the best algorithm. This is the key to writing efficient parallel programs.

Notice that this ability to know when and what communication is incurred is not a property of High Performance Fortran. Recall our earlier discussion of array slices, in which we asserted that it would be reasonable for expression (1) above

$$\dots A[1:n] + B[1:n] \dots$$

not to require communication, but we cannot for several reasons be sure. By contrast, we can be sure that the equivalent ZPL expression

$$[1..n] \quad \dots A + B \dots$$

does not.

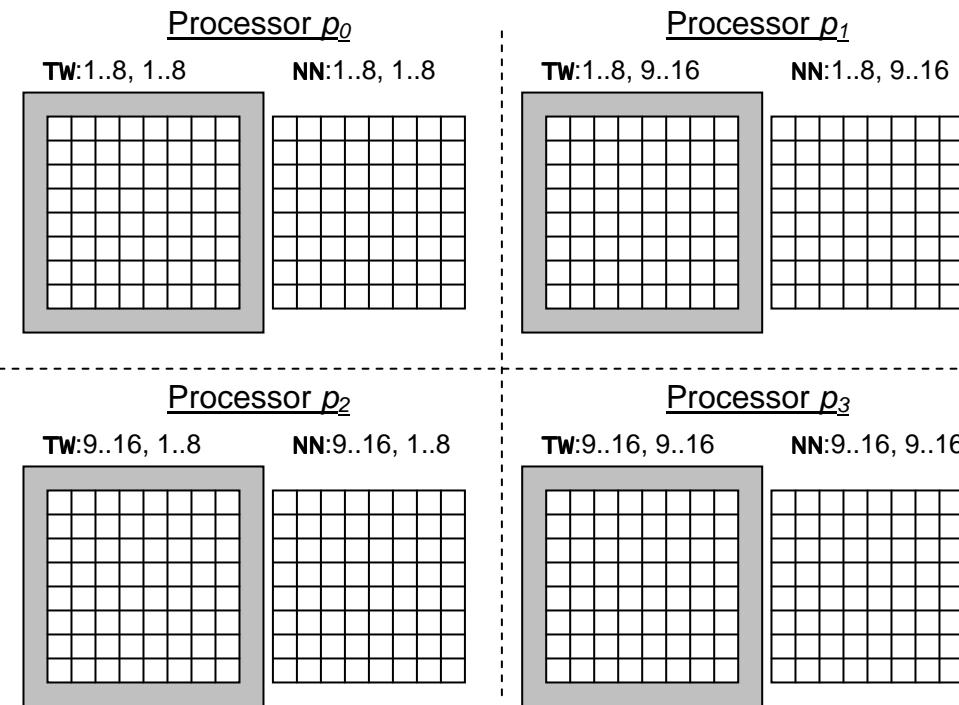


Figure 7. Allocation of the Life program to a 2×2 (logical) processor grid. Notice that array elements having the same index are allocated to the same processors. TW has fluff allocated because some of its references are modified by @-translations; NN does not require fluff.

9.2 Specifics of the Model

To describe the WYSIWYG model, we start on the command line as we invoke a compiled ZPL program. The user specifies along with the computation's configuration values the number of processors available and their arrangement; for our running example, $P=4$, in 2 rows of 2. This is the processor grid, and though in the early years it was thus set for the entire computation, it has since come under programmer control. See Figure 7 for the allocation of a 16×16 problem on the four processors.

During initialization, the regions, and by extension the arrays, are allocated in blocks (by default) to processors, as shown in Figure 7. A key invariant of this allocation is that every element with index $[i, j, \dots, k]$ of every array is always allocated to the same processor (though programmers can override it). So, NN's allocation will match that of TW. This allows all “element-wise” computations to be performed entirely locally. Thus, seeing

$$TW := (TW \& NN = 2) \mid (NN = 3);$$

users know that each processor computes its portion of TW using only the locally stored values of NN and TW. Programmers know they are getting the best performance the CTA has to offer, and the available parallelism ($P=4$) is fully utilized. Such element-wise computation is the sweet spot of parallel computation.

As mentioned, @-communication requires nonlocal data reference. Thus, prior to executing

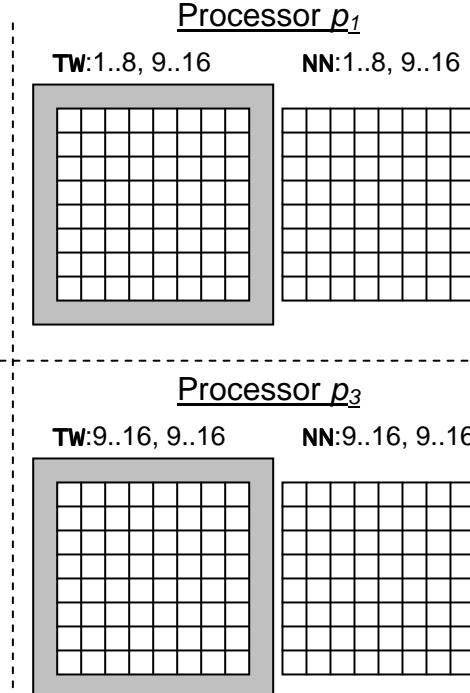
$$\begin{aligned} NN := & TW@^nw + TW@^no + TW@^ne \\ & + TW@^w + TW@^e \\ & + TW@^sw + TW@^so + TW@^se; \end{aligned}$$


Table 1. Sample WYSIWYG Model Information. Work is the amount of computation for the operator measured as implemented in C; P is number of processors. The model is more refined than suggested here.

Syntactic Cue	Example	Parallelism (P)	Communication Cost	Remarks
[R] array ops	[R] ... A+B ...	full; work/ P		
@ array transl.	... A@east ...		1 point-to-point	xmit “surface” only
<< reduction	... +<<A ...	work/ P + log P	2log P point-to-point	fan-in/out trees
<< partial red	... +<<[] A ...	work/ P + log P	log P point-to-point	
scan	... + ...	work/ P + log P	2log P point-to-point	parallel prefix trees
>> flood	... >> [] A...		multicast in dimension	data not replicated
# remap	... A#[I1,I2] ...		2 all-to-all, potentially	general data reorg.

communication is required to fill the fluff buffers with values from the appropriate neighbors. For example, to reference $\text{Tw}@\wedge\text{no}$, the array of north neighbors illustrated in Figure 4, the fluff along the top of the local allocation must contain values from the appropriate neighbor. Figure 8 shows the source of all fluff values required by processor p_0 . Notice from the figure that @-communication does not transmit all values, only those on the “surface” of the allocation—edges and corners—and that d direction references will typically require communication with d other processors. (This example looks especially busy because the instance is so small.) Once a processor has the necessary data, it executes at a speed comparable to the statement without any @-communication.

To conclude the analysis of the Life program, notice that the iteration uses an OR-reduce in the loop control, $!(| << \text{Tw})$. Reduce is another operation that according to the WYSIWYG model entails communication to compute a global result from values allocated over the entire processor grid. The local part of the reduction performs an OR of the locally stored values. The nonlocal part percolates these intermediate values up a combining tree to produce a final result. Because that global result controls the loops executing on each processor, it is broadcast to all processors. These two operations—combine and broadcast—are specified as taking $\log P$ communication steps in the WYSIWYG model, because the combining tree can be implemented on any CTA machine. However, some machines may have hardware for the combining operation as the CM-5 did; alternatively, all processors could fan into a single one if the hardware favored that type of implementation.

The WYSIWYG performance model specifies the characteristics of ZPL’s constructs. Table 1 shows a simplified specification.

The specification of the table is simplified in the sense that true program behavior interacts with allocation, which programmers are fully aware of and control. For example, the programmer *might* have allocated whole columns of an array to the processors, implying there will be no communication in the north/south direction for @ or >> operations. So in the case, when analyzing, say, the SUMMA algorithm in this context, the flood of A’s columns requires a broadcast to all processors, but the flood of B’s rows requires no communication at all. Programmers know all of

this information (they specify it), and so can accurately estimate how their programs will behave.

Because the remap (#) operator can restructure an array into virtually any form, it requires two all-to-all communications, one to set up where the data moves among the processors and one to transmit the data. Remap is (typically) the most expensive operator and the target of aggressive optimizations [41]. Returning to the example above where the elements of a vector were added to their opposite in sequential order,

[1..n] ... A + B#[n-Index1+1] ...

the presence of remap implies by the WYSIWYG model that data motion is necessary to reorder the values relative to their indices. Though two all-to-all communications are specified, the compiler-generated specification for “reverse indices” ($n-\text{Index1+1}$) allows the first of the two to be optimized away; the remaining communication is still potentially expensive. Of course, the actual addition is fully parallel.

The key point is that the WYSIWYG performance properties are a *contract between the compiler and programmers*. They can depend on it. Implementations may be better because of special hardware or aggressive optimizations by the compiler, but they will not be worse. Any surprises in the implementation will be good ones.

9.3 The Contract

To illustrate how the contract between the compiler and the programmer can lead to better programming, return to the topic of slices versus regions, and consider an operation on index sequences of unknown origin, namely

... A[a:b] + B[c:d] ...

Such a situation might apply where there are “leading values” that must be ignored (in B). Because the values of a, b, c and d are unknown, programmers in Fortran 90 will likely use this general solution.

If absolutely nothing is known about the range limits, the ZPL programmer will write

[a..b] ... A + B#[c..d] ...

But using the WYSIWYG model, and knowing that remap is expensive, the programmer might further consider whether there are special cases to exploit. If $a=c$ and $b=d$, then the ZPL

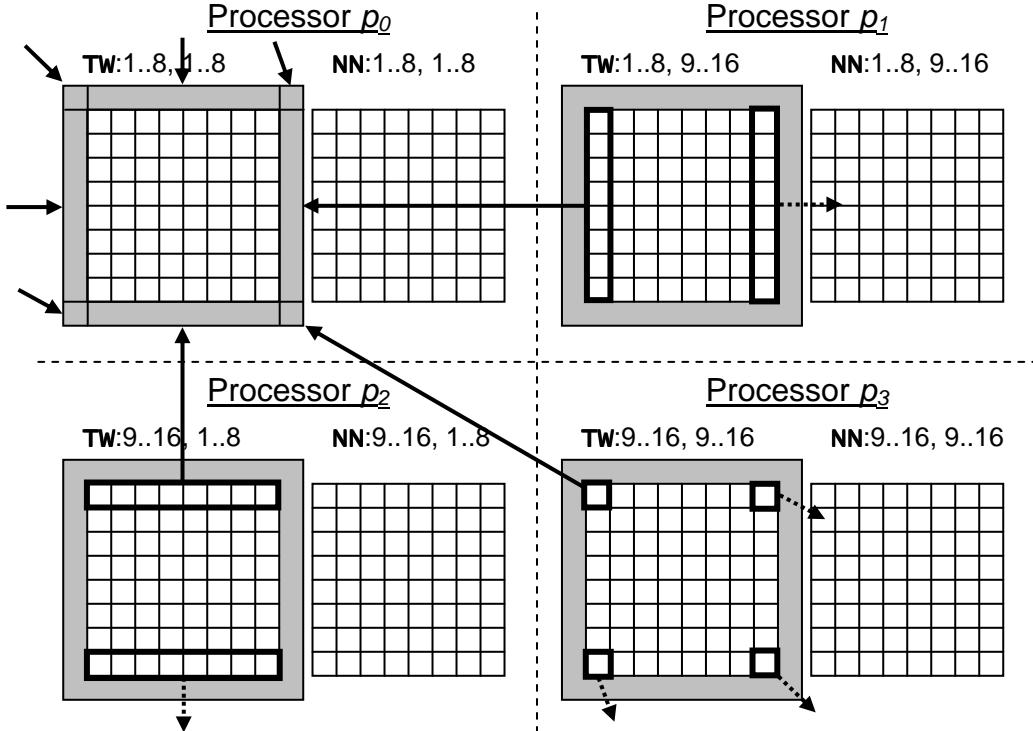


Figure 8. Communication required to load the fluff for **TW on processor p_0 to support wrap @-translations.** Internal array adjacencies are shown by direct arrows from the source values to the appropriate fluff; in all other cases, the reference wraps around the array edge, and the transmission is shown leaving on a dotted arrow and arriving at a solid arrow of the same angle.

[a..b] ... A + B ...

will be an entirely local computation with full parallelism and a great enough “win” that it is worth applying, even occasionally. Further, if the limits are occasionally only “off by 1”, that is, if $c=a+1$ and $d=b+1$, then the ZPL statement

[a..b] ... A + B@right ...

where **right** = [1] is equivalent and also more efficient, despite requiring some communication.

The ZPL compiler generates the exact code required in each case, guaranteed, and because the λ penalty is so large for general communication, it may pay to special-case the situation even if the cases apply in only a fraction of the situations.

9.4 Verifying The Predictions

Creating the WYSIWYG model was easy enough, because it simply describes how the compiler maps source code to the run-time environment coupled with the properties of the CTA. However, its predictive qualities had to be demonstrated. We chose to illustrate the analysis on two well known and highly regarded parallel matrix multiplication algorithms, the SUMMA algorithm [35] and Cannon’s algorithm [42], because they take quite different approaches. (Though we had programmed them often, we hadn’t ever compared them.)

The methodology was to write the “obvious” programs for the two algorithms and then analyze them according to

the dictates of the WYSIWYG model; see Figure 6 for SUMMA and Figure 9 for Cannon. Both computation and communication figure into the analysis.

The analysis divides into two parts: setup and iteration. Only Cannon’s algorithm requires setup: The argument arrays are restructured by incremental rotation to align the elements for the iteration phase. This would seem to handicap Cannon and favor SUMMA, but the situation is not that simple. Ignoring the actual product computation, which is the same for both algorithms, SUMMA’s iteration is more complex because it uses flood rather than @-communication. Floods are implemented with multicast communication, which is generally more expensive than the @-communication. On the other hand, flood arrays have much better caching behavior and the rotation operations of Cannon’s require considerable in-place data motion. So, which is better?

We predicted that SUMMA would be better based on the WYSIWYG model, though completing the analysis here requires a bit more detail about the model than is appropriate [31]. Then we tested the two algorithms on various platforms and confirmed that SUMMA is the consistent winner. Not only was this a satisfying result, but with it we began to realize its value in achieving our goals. *One way to make a language and compiler look good is to empower programmers to write intelligent programs!*

Once we discovered the importance of the WYSIWYG performance model, all subsequent language design was influenced by the goal of keeping the model clear and easy to apply.

```

var A : [1..m,1..n] double; Left operand
    B : [1..n,1..p] double; Right operand
    C : [1..m,1..p] double; Result array
direction right = [0,1]; Ref higher row elements
    below = [1,0]; Ref lower col elements
    ...
    for i := 2 to m do Skew A, cyclically rotate
        [i..m,1..n] A := A@^right; successive rows
    end;
    for i := 2 to p do Skew B, cyclically rotate
        [1..n, i..p] B := B@^below; successive columns
    end;
[1..m, 1..p] C := 0.0; Initialize C
    for i := 1 to n do For common dimension
        [1..m, 1..p] C += A*B; Accumulate product terms
        [1..m, 1..n] A := A@^right; Rotate A
        [1..n, 1..p] B := B@^below; Rotate B
    end;

```

Figure 9. Cannon’s matrix multiplication algorithm in ZPL. To begin, the argument arrays are skewed so rows of A are cyclically offset one index position to the right of the row above, and the columns of B are cyclically offset one index position below the column to its left; in the body both argument arrays are rotated one position in the same directions to realign the elements.

10. Compiler Milestones

The ZPL compiler uses a custom scanner with YACC as its front end, and we initially relied on symbol table and AST facilities from the Parafrase compiler. (As the compiler matured the Parafrase code was steadily eliminated in favor of custom ZPL features; it is not known how much Parafrase code remains, if any.) As new language features were added to ZPL, the corresponding new compiler technology was usually implemented by additional passes over the AST. A final code generation pass emitted ANSI C to compete the compilation process.¹⁴ The overall structure is shown in Figure 10.

Several aspects of the compiler are significant.

10.1 Ironman – The Compiler Writer’s Communication Interface

A compiler that seeks to generate machine-independent parallel code must decide how to handle interprocessor communication, which is as particular to a computer as its instruction set. Standard compiler theory suggests solving the problem by targeting machine independent libraries or producing a separate backend for each platform. The latter is time consuming (and platforms come and go with astonishing rapidity), so writers of parallel language compilers target the lowest common denominator: the machine independent message-passing library. All parallel computer communication can be reduced to message passing. Two

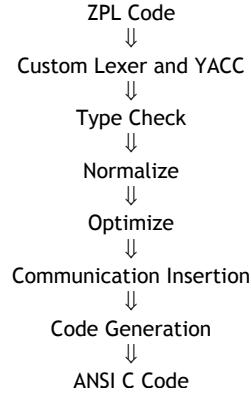


Figure 10. Principal components of the ZPL compiler.

libraries are in wide use: Parallel Virtual Machine (PVM) [42] and Message Passing Interface (MPI) [43]. But, in our view compiling to the message passing level introduces too much “distance” between compiled code and the computer.

Specifically, message passing is a communication protocol with a variety of forms, ranging from a heavyweight synchronous form with multiple copying steps, to lighter-weight asynchronous forms. Without delving into details, all schemes have a sender that initiates the communication (`send`) and must wait until the data is “gone,” and a receiver that explicitly receives it (`recv`) when it “arrives” to complete the transmission; further, when noncontiguous data addresses are sent—think of a column of data for an `A@west` transmission in a row-major-order allocation—data must be marshaled into a message, i.e. brought together into a separate buffer. But there are other kinds of communication: In shared-memory computers data is transmitted by the standard load/store instructions; other machines have “one-sided” get/put communication. Both schemes are extremely lightweight compared to message passing, e.g. neither requires marshalling. Message passing is too thick a layer as the one-size-fits-all solution.

The ZPL commitment to targeting all parallel computers made us aware of these differences early. We had originally generated *ad hoc* message passing calls, but then realized that we needed a way to generate generic code that could map to the idiosyncrasies of any target machine. Brad Chamberlain with Sung-Eun Choi developed the Ironman interface, a compiler-friendly set of communication calls that were mechanism independent [44]. Unlike message-passing commands, the Ironman operations are designed for compilers rather than people.

Ironman conceptualizes interprocessor communication as an assignment from the source to the target memory, and the four Ironman calls are the def/use protocol bounding the interval of transmission (see Figure 11):

- `destination_ready()` marks the place where the buffer space (fluff) that will be written into has had its last use
- `source_ready()` marks the place where the data to be transmitted has been defined
- `destination_needed()` marks the place where computations involving the transmitted data are about to be performed – impending use

¹⁴ One of our first and most patient users was an MIT physicist, Alexander Wagner, who used ZPL for electromagnetic modeling [69]. His was a complex 3D simulation with large blocks of very similar code. Ironically, he wrote a C program to generate the ZPL for the blocks, which we then compiled back to C!

- `source_volatile()` marks the place where the data is about to be overwritten – impending def

The procedures' parameters say what data is involved and name the destination/source memory. Pairs of calls are placed inline wherever communication is needed.

This compiler-friendly mechanism specifies the interval during which transmission must take place, but it subordinates all of the details of the transmission to whatever implementation is appropriate for the underlying machine. The appropriate Ironman library is included at link time to give a custom communication implementation for the target machine.

Table 2 shows bindings for various communication methods. Notice that the four Ironman routines serve different functions for different methods. For example, in message-passing protocols that copy their data, the `DR()` and `SV()` functions are no-ops, because when the data is ready to be sent it is copied from the source, allowing the source to be immediately overwritten; when the receive operation is performed, it must wait if the data has not arrived.

One-sided and shared-memory implementations of Ironman can transfer the data immediately after it is available, that is, in `SR()`, or just before it is needed in `DN()`. The former is preferred because if the optimizer is able to find instructions to perform on the destination processor, that is, $k > 0$ in Figure 11, then communication will overlap with computation. Such cases are ideal because they hide the time to communicate beneath the time to compute, eliminating (some portion of) the time cost for that communication.

Developing these optimizations was the dissertation research of Sung-Eun Choi [45]. Although there are numer-

ous variations, one underlying principle is to spread the interval over which communication could take place; that is, move `destination_ready()` and `source_ready()` earlier in the code, and `destination_needed()` and `source_volatile()` later, causing k to increase. This usually improves the chances that communication will overlap in time with computation, hiding its communication overhead. (She also implemented a series of other optimizations including combining, prefetching and pipelining [45].)

The proof that the message passing abstraction is “too distant” was nicely demonstrated on the Cray T3E, a computer with one-sided (Shmem) communication. A ZPL program was written and compiled. That single compiled program was loaded and run with ZPL’s Ironman library implemented with Shmem (get/put) primitives and with the Ironman library implemented with Cray’s own (optimized) version of MPI (send/recv) functions. The one-sided version was significantly faster [44], implying that a compiler that reduces communication to the lowest common denominator of message passing potentially gives up performance.

10.2 The Power of Testing

The ZPL compiler development was conducted like other academic software development efforts. All programmers were graduate students who were smart, well schooled compiler writers; we used available tools like RCS and CVS, and we adopted high coding standards. But, like all research software, the goal changed every week as the language design proceeded. The emphasis was less on producing robust software and more on keeping up with the language changes. This meant that the compiler was an extremely dynamic piece of software, and as it changed it

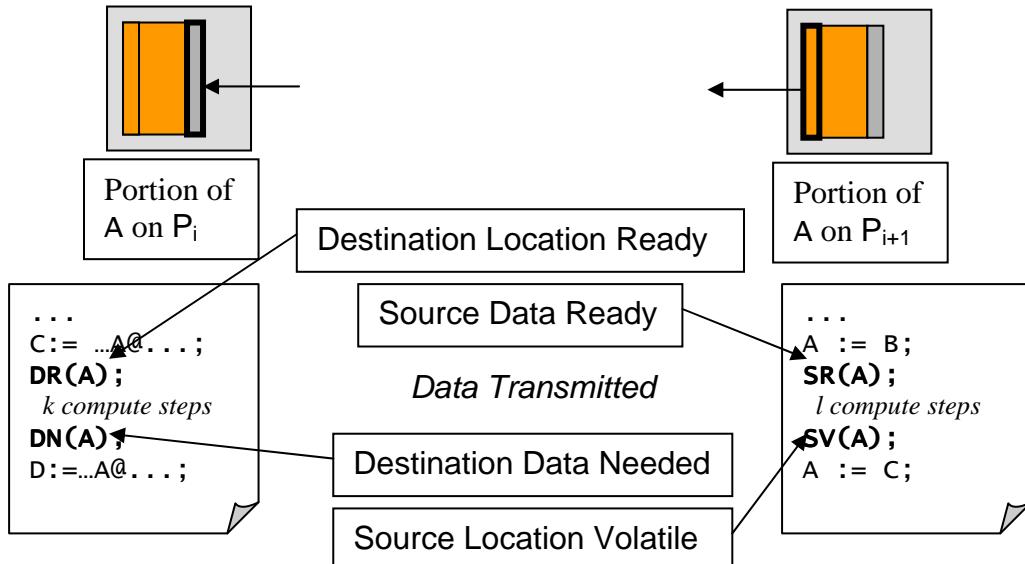


Figure 11. Schematic of Ironman call usage for a single transmission. As a result of an `A@east` operand reference, a column of `A` is transmitted from P_{i+1} , the source, to P_i , the destination. (Because ZPL generates SPMD code, P_i will also be a source, i.e. include `SR()` and `SV()` at this point, and P_{i+1} will also be a destination, i.e. include `DR()` and `DN()` at this point, to other transmissions.) The four Ironman calls are inserted by the compiler into the program text.

accreted: Features were added, then eliminated, leaving dead code, or worse, nearly dead code; implementers, uncertain of all interactions among routines, would clone code to support a new feature, leaving two very similar routines performing separate tasks, etc. And the compiler was fragile for a variety of reasons, mostly having to do with incomplete testing.

Eventually, perhaps in late 1995 or early 1996, Jason Secosky, who was taking the software engineering class, decided ZPL needed a testing system, and he built one. The facility woke up every night and ran the entire suite of ZPL test programs, reporting the next morning to the team on which of these regression tests failed.

Immediately, the testing system resulted in two significant changes to the compiler. First, the programmers became bolder in their modifications to the compiler. It was possible to make changes to the code and if something broke, it would be announced the next morning in complete detail. Second, the compiler began to shrink. Procedures with common logic were removed and replaced by a more general and more robust routines. Very quickly, the compiler tightened up.

Though the worth of continual testing is well known, it is easy to put off applying the tool too long. So significant was the improvement in the ZPL compiler that we all became evangelists for regression testing.

10.3 Scalar Performance

ZPL is a parallel language, the beneficiary of a long list of failed attempts to create an effective parallel programming system. As described above, our focus was on generating high-quality parallel code. By 1996 we were generating quality code and had been demonstrating performance with portability for some time. We were not, however, doing well enough (in our opinion) against Fortran + MPI benchmarks, and the problem was not with the parallel part of our generated code, but with the scalar portion. We were not generating naïve code, but we were not pulling out all stops either. Though we had knocked off the parallel compilation problem, it is a fact that virtually all of the “parallel part” is (necessary) overhead relative to the scalar part that actually does the work. We spent a year improving programs, especially for scalar performance.

One simple example was the Bumpers and Walkers optimization, a mechanism for walking an array using pointer arithmetic rather than direct index calculations. Bumpers move through successive elements, walkers advance through higher dimensions. Though it was somewhat subtle

to get the logic fully general considering that ZPL has fluff buffers embedded in arrays used in @ references, the improvement was significant given how common array indexing is in ZPL.

A large list of optimizations focused on applying standard compiler techniques in the ZPL setting. For example, eliminating temporaries is a tiny optimization in the scalar world, but in the ZPL world temporaries are arrays; removing them can be significant. Temporaries are introduced automatically whenever the compiler sees the left-hand side operand on the right-hand side in an @-translation, as in

```
[R] A := A@west;
```

This specific example, performed in place, would overwrite itself using the standard row traversal indexing from 1 to n , forcing the temporary. However, no temporary is needed if the references proceed in the opposite direction, n to 1. A compiler pass can analyze the directions to determine when temporaries can be eliminated. Because the compiler does not distinguish between whether the temporary was placed by the user or the compiler, it usually finds user temporaries to eliminate.

Another very effective improvement for ZPL’s scalar code is loop-fusion. Compilers recognize when two consecutive loops have the same bounds and stride, and combine them into a single loop in an optimization known as **loop-fusion**. Fusing not only reduces loop control overhead and produces longer sequences of straight line code in the loop body, but it can, if the loops are referencing similar sets of variables, improve data reuse in the cache. (The reason is that in the double-loop case a value referenced in the first loop may have been evicted from the cache by the time it is referenced in the second loop, though the situation is quite complex [46].) ZPL’s loop fusion research, chiefly by E Chris Lewis, advanced the state of the art on the general subject and improved ZPL code substantially. His work also advanced **array contraction**, an optimization in which an array that cannot be eliminated is reduced in size, usually to a scalar. Replacing an array with a scalar reduces the memory footprint of the computation, leading both to better cache performance and to the opportunity to solve larger problems [47].

Yet another class of improvements, the dissertation research of Derrick Weathersby [48], involved optimizing reductions and scans. Reduction operations ($op \ll$) are often used on the same data and in tight proximity, as in

Table 2. **Sample bindings to the four Ironman procedures.** Standard communication facilities are used to implement the four Ironman routines.

	Copying message passing (nCUBE, iPSC)	Asynchronous message passing MPI Asynch	Put-based 1-sided communication (Cray T3D, T3E)	Shared memory with coherency, SMPs
Destination Loc Ready		mpi_irecv()	post_ready	post_ready
Source Data Ready	csend()	mpi_isend()	wait_ready shmem_put post_done	wait_ready fluff:= A post_done
Destination Data Needed	crecf()	mpi_wait()	wait_done	wait_done
Source Volatile		mpi_wait()		

```
[R] range := (max<< A) - (min<< A);
```

Such cases are ideal cases for merging. A reduction usually involves three parts: combining local data on each processor in parallel, a fan-in tree to combine the P intermediate results, and a broadcast tree notifying all P processors of the final result. The last two steps involve point-to-point communications of single values to implement a tournament tree. Since two or a small number of values fit in the payload of most communication facilities, that is, a few values can be transmitted together at the cost of transmitting a single one, it makes sense to merge the fan-in/fan-out parts of reductions to amortize the cost over several operations [48].

One other group of important optimizations included Maria Gulickson's array blocking optimization to improve cache performance, Steven Deitz's stencil optimizations [49] to eliminate redundant computation in the context of @-references, and Douglas Low's partial contraction optimizations to reduce the size of array temporaries when they cannot be removed or reduced to a scalar.

The efforts were essential and rewarding—it's always satisfying to watch execution time fall—but it also seemed to be slightly orthogonal to our main mission of studying parallel computing.¹⁵

10.4 Factor-Join Compilation

In the same way that the WYSIWYG model called the user's attention to communication in the parallel program, it called our attention to the interactions of communication and local computation, too. For example, recognizing adjacent uses of the same communication structure usually offered an opportunity to combine them at a substantial savings, and “straight-line” computation between communication operations defined basic blocks that were prime opportunities for scalar optimization. This motivated us to abstract the compilation process in terms we described as Factor-Join [50].

In brief the idea is to represent the computation as a sequence of factors—elements with common communication patterns—and then join adjacent factors as an optimization. The approach was not a *canonical form* like Abrams' APL optimizations (Beating and Drag-along [51]); ZPL probably doesn't admit a canonical form because of the distributed nature of the evaluation. But Factor-Join regularized certain communication optimizations in such a way that they could in principle be largely automated. Exploiting these ideas would be wise when building the next ZPL

compiler, but our compiler was never retooled to incorporate these ideas cleanly.

10.5 Going Public

Finally, in July 1997 ZPL was released. Though there was much more we wanted to do, the release was a significant milestone for the team. Users had worked with the language for years, so we were confident that it was a useful tool for an interesting set of applications. It might not be finalized, but it was useful.

There was a ripple of interest in ZPL, but not a lot. We had anticipated that the release would draw only modest interest for a set of obvious reasons: First, users are notoriously reluctant to adopt a new language; they are even less likely to adopt a university-produced language; and we were unabashedly a research language, which users had to expect would change and have bugs. Second, we were releasing binaries, not the sources, and they were targeted to the popular but not all extant parallel machines. Third, the software came as is, with no promises of user support or 800 number.¹⁶ Finally, the entire community was focused on High Performance Fortran.

I don't know if the team was disappointed in the response or not—it never came up. But I was happy for the interest we got and relieved it was no greater. We didn't have the funding to provide user support, and I could not use the team for that purpose beyond our current level of expecting team members to be responsible for fixing their own bugs. Further, much more remained to do, and I wanted to preserve the option of changing the language. For the moment, we only needed enough users to keep us honest, and no more. We had that many. Over the next few years while we worked on the language, we made no effort to popularize ZPL, though users regularly found us and grabbed a copy. Periodically, we would upgrade the public software, but that was all.

11. After Classic ZPL

The language released has become known as ZPL Classic, but at the time we saw ourselves moving on to Advanced ZPL, which we abbreviated A-ZPL to suggest its generality. Indeed, we were sensitive to the fact that although ZPL Classic performed extremely well for the applications that exploited its abstractions, these abstractions needed to be more flexible and more dynamic. We'd tamed regular data parallelism; now it was time to be more expressive.

Though I'm not aware of any study proving the effect, I believe that during the five years of ZPL development 1992-1997, there was a significant advancement in parallel algorithms for scientific computation: They became much more sophisticated. If true, it was likely the result of substantial funding increases (High Performance Computing Initiative) and collaboration between the scientific computing community and computer scientists. The bottom line for us was that although ZPL Classic was expressive enough for 1992-vintage algorithms, 1997 algorithms wanted more flexibility. We were eager to add it.

¹⁵ The effort was vindicated several years later by George Forman, a founding ZPL project member who had graduated years earlier and was working on clustering algorithms at Hewlett-Packard. While gathering numbers for a paper on performance improvement through parallelism, he wanted to compare their ZPL program to a sequential C program they had written during algorithm creation. Their measurements showed that ZPL's C “object code” was running significantly faster than their C program on one processor. George called to complain about our timer routines, but in the end, the numbers were right. The aggressive scalar optimizations were the reason. ZPL simply produces tighter, lower-level (uglier?) C code than humans are willing to write, perhaps making ZPL a decent sequential programming language.

¹⁶ Nevertheless, the team took it as a matter of pride to respond rapidly to users' email.

This section enumerates important additions to the language.

11.1 Hierarchical Arrays

Multigrid techniques are an excellent example of a numerical method that became widely popular in the 1992-97 period, though it was known before then [52]. A typical multigrid program uses a hierarchy of arrays, the lowest level being the problem state; each level above has 1/4 as many elements in the 2D case as the array below it; a single iteration “goes up the hierarchy” projecting the computation on the coarser grids, and then it “goes down the hierarchy” interpolating the results; convergence rates are improved. For ZPL, hierarchical arrays greatly complicated regions, directions and their interactions.

Renumbering In an array programming language, multigrid computations require manipulating arrays with different, but related, index sets. The significant aspect of supporting multigrid is whether the indices are renumbered with each level. Figure 12 shows a 2D example illustrating the two obvious cases: not to renumber (12(a)), ZPL’s choice, and to renumber (12(b)). That is, assuming the lower array has dense indices $[1..8, 1..8]$, not renumbering implies that the level above has sparse indices $[1..8 \text{ by } 2, 1..8 \text{ by } 2]$, making the second element in its first row (1,3); renumbering implies that the level above has dense indices $[1..4, 1..4]$, making the second element in its first row (1,2).

We struggled with the decision of whether to renumber for some time. Our colleagues in applied math who were eager to use the abstraction were initially stunned when we asked about renumbering indices. They never thought about it, and simply assumed the indices were dense at every level. For us, it seemed natural to express projecting up to the next level by the intuitive (to ZPL programmers) statement

```
[1..n, 1..n by [2,2]] AUP :=
    (A+A@east+A@se+A@southeast)/4;
```

which would imply no index renumbering. Though the “more intuitive” argument was strong, it was nowhere near as compelling as the WYSIWYG argument.

In ZPL an index position describes the allocation of its value completely, allowing the WYSIWYG model to describe when and how much communication takes place. For example, in Figure 12 the WYSIWYG model tells us the communication required for each scheme: In Figure 12(a) no communication is required to project up the hierarchy; in Figure 12(b) there is communication between the bottom level and the level above, though not for the next level; exact details depend on the stencil, sizes and processor assignment. In general, renumbering will force considerable communication at every level until the entire renumbered array fits on one processor. Further, the WYSIWYG model tells us that the load is balanced in Figure 12(a) but unbalanced in 12(b). The issues are somewhat more complicated “going down,” but the point is clear: better performance can be expected with no renumbering.

The decision to not to renumber has stood the test of time: I remain convinced that it is the only rational way to formulate multigrid in a region-based language. We pointed out to our colleagues from Applied Math that our decision hardly mattered to them, because indices are so rarely used once the regions are set up correctly: They

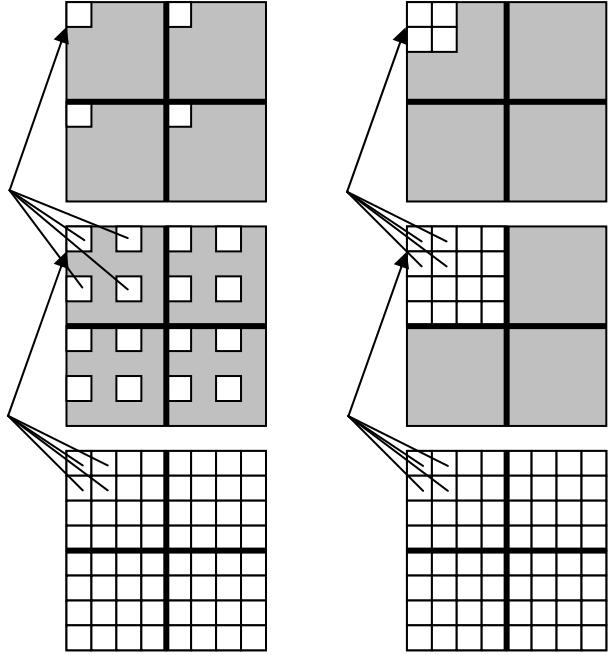


Figure 12. Multigrid. Three levels of a hierarchical array are used to express a multigrid computation; the index positions are preserved and the (logical) size remains constant in (a); the index positions are renumbered and the footprint of the array shrinks in (b).

could continue to think of arrays as being renumbered as they programmed the computation. The actual indices matter only when analyzing performance, and if for some reason it would be better to make the arrays dense, a remap solves the problem.

Curly Braces Multigrid was on the cusp between ZPL and A-ZPL. Reviewing what we’d created as we moved on to A-ZPL, we were pleased that we’d figured out how to incorporate array abstractions for multigrid computations into an array language; it was surely a first. Further, the abstraction was compatible with WYSIWYG, also a major accomplishment. We were getting good performance on the NAS MG benchmark with many fewer lines of code; see Figure 13. And our first user, Joe Nichols, a UW mechanical engineering graduate student, got his combustion program to work without massive amounts of help from us; it was also faster and shorter than the C + MPI equivalent.

But there was much to criticize. ZPL programs had always tended to be clean, but multigrid computations were not. Our approach for specifying multi-regions, arrays and directions, which must get progressively sparser, was to parameterize them by sequences in curly braces. Without going into the details, the arrays for Figure 12(a) and 12(b) would have the form

```
region
  Fig12A{0..2} =
    [1..8,1..8] by [2^{},2^{}]; strides 1, 2, 4
  Fig12B{0..2} =
    [1..8/(2^{}),1..8/(2^{})]; renumbered
var   A{}: [Fig12A{}] float;
```

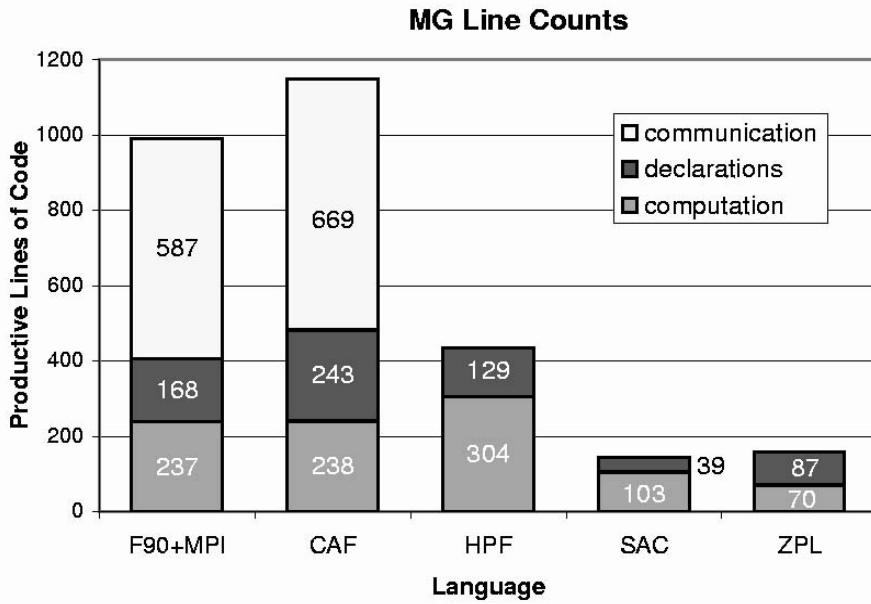


Figure 13. Line Counts for the NAS Multigrid Benchmark [63]. Lines after removing comments and white space are classified into user-specified communication, declarations or computation; published data for Fortran-90 using the Message Passing Interface, Co-Array Fortran (restricted at the time to Cray Shmem machines) [70], High Performance Fortran, Single Assignment C (restricted to shared memory computers) [59] and ZPL.

```
B{}: [Fig12B{}] float;
```

which was decidedly not beautiful. Curly braces were epidemic in all expressions involving hierarchical arrays, as this (3D) projection code:

```
procedure rprj3(var S,R: [, , ] double);
begin
  S:=0.5000 * R +
    0.2500*(R@^dir100{}+R@^dir010{}+R@^dir001{}) +
    R@^dirN00{}+R@^dir0N0{}+R@^dir00N{})+
  0.1250*(R@^dir110{}+R@^dir101{}+R@^dir011{}) +
    R@^dir1N0{}+R@^dir10N{}+R@^dir01N{}) +
    R@^dirN10{}+R@^dirN01{}+R@^dir0N1{}) +
    R@^dirNN0{}+R@^dirN0N{}+R@^dir0NN{})+
  0.0625*(R@^dir111{}+R@^dir11N{})+
    R@^dir1N1{}+R@^dir1NN{})+
    R@^dirN11{}+R@^dirN1N{})+
    R@^dirNN1{}+R@^dirNNN{}) ;
end;
```

from the NAS MG illustrates. Further, as discussions with Nichols and users from Applied Math indicated, parameterized regions and directions were not intuitive.

Over time, we came to this conclusion [33]: The complications of this hierarchical array formulation with its parameterization were inevitable because ZPL's regions and directions were not first class. To be **first class** means they should be treated like any other value. (The fact that ZPL's regions and directions were not first class was an oft-criticized aspect of ZPL Classic.) We had generalized them to support hierarchical structures, but this was too situation-specific. If we made regions and directions first class—thereby permitting arrays of regions and arrays of directions—then a clear, orthogonal formulation of hierarchical arrays with all the desirable WYSIWYG properties would fall out. And the cumbersome direction references

(also evident in the foregoing code) would be fixed. Chamberlain presented this analysis in his dissertation [33], and that is how multigrid is now computed in A-ZPL.

11.2 User-defined Scans and Reduces

Scan is a parallel prefix operator, which, like reduce, uses the base list of associative operators (+, *, &, |, max, min); unlike reduce, scan returns all intermediate results. ZPL always included scan, but few numerical programmers use it.¹⁷ Nevertheless, researchers like Blelloch [53] have argued that scanning is a powerful parallel problem solving technique. We agree and were motivated to make it a powerful tool in ZPL.

The idea of user-defined reductions and scans was first discussed in the early days of APL. They have been implemented sequentially, and on shared-memory parallel machines, but the full application of the parallel prefix algorithm [54] on the CTA architecture remained unsolved. The problem was that in the standard sequential implementation the reduce/scan proceeds through the elements first to last, processing one result at a time directly. In the ZPL formulation of the parallel prefix algorithm each processor performs a local reduce/scan, and then these local intermediate results are combined; finally, for the scan the local values must be updated based on intermediate results computed on other processors. Steven Deitz worked out the logic for generalized reduction and scan [55, 56]. Our algo-

¹⁷ The exception proving the rule was a solar system simulation testing planetary stability written with UW Astronomers by Sung-Eun Choi; the inner loop was a scan over records each of which was a descriptor of the orbits of the planets for an (Earth) year.

rithm must be broken into five parts: initialize, local, combine, update (for scan only) and output.

Further, the input and output data types processed by these routines are potentially different. For example, the `Third_Largest_Value` reduce of a numeric array is an example of a user-defined reduction; it returns the input array's third largest value. The base input data type is numeric, but the intermediate routines pass around records of numeric triples, and the output is again numeric. The four components for this reduction have the following characteristics:

- Initialize starts the local reduction on each processor, emitting a record triple containing identity values.
- Local takes a numeric value and a triple and returns a new updated next triple.
- Combine takes two numeric triples and returns the triple of their three largest numbers.
- Output takes the global triple and returns the final numeric result.

User defined reductions and scans enjoy the same benefits (mentioned earlier) as their built-in counterparts, such as participating in combining optimizations.

As a postscript, notice two curious properties of scan.

- As mentioned earlier, ZPL started out following APL by specifying that partial reduce and scan use dimension numbers in brackets, as in the APL, `+/[2]A`. Partial reduction was changed to use the “two regions” solution as a result of formulating flood, but partial scan continues to specify the dimensions by number, as in

```
[R] Temp := +||[-2] A;
```

where negative numbers indicate that the scan is to be performed from high indices to low. The key reason for the apparently inconsistent syntax for apparently similar operations is that scan doesn't change rank.¹⁸

- A moment's thought reveals that the compiled code for the procedure calls and interprocessor communication for higher-dimensional scans is potentially very complex. But as several researchers have noted, higher-dimensional scans can be expressed in terms of a composition of lower-dimensional scans that can be implemented in a source-to-source translation. Deitz [55] observed that the key to this technique in the context of user-defined scans is to choose exclusive rather than inclusive scans. The difference is whether each element is included in its replacement value or not, as in

$$\begin{array}{lll} +|| \ 2 \ 4 \ 6 & \Leftrightarrow & 2 \ 6 \ 12 \\ +|| \ 2 \ 4 \ 6 & \Leftrightarrow & 0 \ 2 \ 6 \end{array} \quad \begin{array}{l} \text{Inclusive} \\ \text{Exclusive} \end{array}$$

where the first element in the exclusive scan is the identity for the operation. The inclusive form can be

found from the exclusive by element-wise combining with the operand. Though APL and most languages use the Inclusive form, A-ZPL ultimately adopted the more general exclusive scan, because generating code for high-dimension user defined scans is extremely difficult otherwise.

Scan is an operation worthy of much wider application.

11.3 Pipelines and Wavefronts

It is not uncommon in scientific computation to compute values that depend on just computed values elsewhere in the array. For example, the new value of $A_{i+1,j+1}$ may depend on $A_{i,j}$, $A_{i+1,j}$ and $A_{i,j+1}$. This looks more like a scan in two dimensions (simultaneously) than it looks like the other whole-array operations of ZPL. But it is a bigger and more complex scan. Our solution quickly took on the name **mighty scan**.

Whereas standard scan and its user-defined generalizations apply to array elements in one dimension (at a time) with a single update per element, wavefront computations must proceed in multiple dimensions at once (three is most typical) and perform much more complex computations. The solution, developed by E Chris Lewis [47], involved two additions to the language: *Prime ats* and *scan blocks*.

The @-translations were extended to add a prime, as in

```
[1..n, 2..n] A := A + A'@west;
```

The **prime at** indicates that the referenced elements refer to the *updated* values from earlier iterations in the evaluation of *this* statement. Thus, in the illustrated statement, the first column of A would be unchanged (note region), the second column would be the sum of the first two, the third would be the sum of the first three, because it combines the third column (A) and the newly updated second column A'@west, etc. (Of course, this simple one-dimension example can also be expressed as a partial scan.)

Production applications perform complex calculations in wavefront sweeps, updating several variables at once. Sweep3D, a standard wavefront benchmark, updates five 3D arrays, four as wavefronts, in its inner loop. In order to permit such complex wavefronts, the scan block was added as a statement grouping construct

```
[R] scan
    A := A*B + A@w*B@w + A@nw*B@nw + A@n*B@n;
    B := r*B + s*B@w + t*B@nw + u*B@n;
    end;
```

Essentially, the scan block fuses the loop nests of all the array assignments into a single loop nest, allowing the updates to be propagated quickly [57]. Notice that because these operations are performed in parallel on values resident on other processors, only a subset of the processors can start computing at first. Once edge data can be sent, adjacent processors can begin computation. This raises interesting scheduling questions relative to releasing blocked processors quickly.

¹⁸ Notice the three related operators reduce (*op* <>), scan (*op* ||) and flood (>>) have the properties that the result is smaller than, the same as, or larger than the operand, motivating the choice of operator syntax.

The prime at and scan block were extremely successful in the parallel benchmark Sweep3D.¹⁹ The core of the Sweep3D Fortran code is 115 lines, while it is only 24 in ZPL. Further, ZPL’s performance is competitive (+/- 15%) with the Fortran + MPI on the measured platforms [47].

11.4 Sparse Regions and Arrays

ZPL Classic has dense regions, strided regions, and regions with flood dimensions. Arrays can be defined for all of them. Further, it has a `with` region operator for masking. But, as our numerical computation friends kept reminding us, most huge problems are not dense. Brad Chamberlain was especially concerned about the matter, and ultimately cracked the problem. There were several issues, and since MATLAB was the only language offering sparse arrays—but interpreted sequentially rather than compiled in parallel—he had a lot of details to work out.

The first concerned **orthogonality**, meaning that because exploiting sparseness is simply an optimization, the use of sparse arrays and regions should match the use of ZPL’s other forms of arrays and regions, differing possibly only by the `sparse` modifier on the declarations. Chamberlain’s formulation achieved that, but there is a wrinkle to be explained momentarily.

The second concerned performance. Computer scientists and users have created a multitude of sparse array representations, and each has its advantages. We wanted a solution that worked well—defined to mean that the overhead per element was a (small) constant for all sparse configurations. Further, the representation must work well in ZPL, meaning that it fits within the specifications of the WYSIWYG model. Since there is no universal sparse representation, the internal representation was a difficult task; our one advantage was that the operators and data-reference patterns of the rest of the language were known, and they were the only ones that could manipulate sparse arrays.

The third issue was, what does “sparse” mean? Numerical analysts think of a sparse array as containing a small number of nonzero elements, where “small” is in the eye of the beholder, but is surely a value substantially less than half. “Zero” means a double precision floating-point 0. But couldn’t a galaxy photo be a sparse pixel array where “zero” means “black pixel”, i.e. an RGB triple of zeroes? We agreed that a sparse array was any array in which more than half of the elements were *implicitly represented values* (IRV), and the user can define the IRV.

The unexpectedly difficult aspect of sparse arrays, the wrinkle, concerned initialization. The issue is that, given a dense array, it is easy to produce a sparse region from it efficiently, say by masking (`with`). And given a sparse region, it is easy to produce another sparse region from it. But how to create a huge sparse region in the first place without creating a huge dense region to cover it? It could be read in, but in ZPL I/O must be executed in some region context, which will either be a dense region or lead to a circular solution. Another approach is to create the initial

sparse region by parallel computation, say with a random number generator, but how is this computation specified if the region does not yet exist? In the end the general problem was never solved. ZPL has full support for sparse regions, but they are either created from dense regions or set up with tools external to the language. This latter is not wholly unsatisfactory, because sparsity patterns often have odd sources that require external support to use.

Chamberlain advanced the state of the art dramatically, producing the first array language abstraction and first comprehensive compiler support for sparse arrays. It is unquestionably one of the premiere accomplishments of the ZPL research. More remains to be done: the sparse array initialization problem is a deep and challenging topic worthy of further research.

11.5 Data Parallelism and Task Parallelism

ZPL Classic is a data-parallel language, and A-ZPL adds features that go well beyond data parallelism. But it is widely believed that task parallelism is also an essential programming tool, and we were eager to include such facilities in A-ZPL. Several data-parallel languages have been generalized to include task parallelism, and vice versa. Steven Deitz investigated which of these was the most successful, and concluded that none had been. Essentially, his analysis seemed to imply that once a language acquires its “character”—data parallelism or task parallelism—fitting facilities for the other paradigm into it cleanly and parsimoniously is difficult, we guessed impossible. This result was not good news.

Nevertheless, Deitz continued to investigate how to add more flexibility and dynamism in ZPL’s assignment of work and data to processors. Of several driving applications, the adaptive mesh refinement (AMR) computation was one he considered closely.

To appreciate the solution, recall that the parallel execution environment of ZPL is (by default) initially defined in an extra-language way. That is, the number of processors and their arrangement are specified on the command line. Parameters (configuration constants and variables) are also defined on the command line, and are used to set up regions and arrays when the computation starts. These inputs define the default allocation of arrays to processors, which is the basis of understanding the WYSIWYG performance model. ZPL is always initialized this way, and previously, none of it could be changed in the program.

New Features To add flexibility two new abstractions were added to the language, *grids* and *distributions* [55]. They are both first class, and with regions elevated to first-class status, programmers can completely control all the features originally provided by default.

Grids are a language mechanism for defining the arrangement of processors. The values of grid names are dense arrays of distinct processor numbers 0 to $P-1$ of any rank. The assignment of numbers to positions is under user control, giving considerable flexibility in arranging logical processor adjacencies. In concept, the initial (command line) setup defines P and sets the first arrangement.

Distributions are a mechanism for mapping regions to grids. The user specifies the range of indices to be assigned in each dimension. Further, the type of allocation scheme—block, cyclic, etc.—is also specified for each dimension. Users can default to compiler-provided implementation routines for the allocation, or program their own. Again,

¹⁹ Indeed `scan/end` are sufficiently powerful that they have been generalized and renamed as `interleave/end`.

the initial setup defines the initial (block) assignment of the regions.

The typical use of these facilities to redefine the runtime environment is as follows. Within the current parallel context, determine how the data should be reallocated. Then

- ↓ Restructure the processors into a new grid, if necessary,
- ↓ Specify how the data should be allocated in the new arrangement, if necessary,
- ↓ Define a new region to assign appropriate indices to the problem, if necessary, and
- ↓ Say whether the array reallocation is to preserve or destroy the data contained therein.

If these operations are specified in consecutive statements, they are bundled together and performed as one “instantaneous” change in environment. This strategy minimizes data motion and the use of temporaries. The statements can realize a complete restructuring of the computational setting.

Notice that although there is considerable flexibility, one feature—that all arrays with the same distribution and a given index, say i, j, \dots, k , will have that element allocated to the same processor—is maintained by these ZPL facilities. This is a property of the base WYSIWYG model, and preserving it is key to preserving the programmer’s sanity. After all, once grid-distribution-region-array operations set up a new parallel context, the user must morph his or her understanding of the WYSIWYG model to accord with it. This isn’t really difficult, because the WYSIWYG ideas were probably used to create the new context in the first place. Nevertheless, treating the indices consistently helps.

In Place Restructuring The distribution facilities are extremely interesting and worthy of much greater study. However, there are two curious aspects worth mentioning.

When a distribution binds a region to a grid, there is a question of what happens to the data stored in the arrays when that region had a different structure. It could be lost, or it could be preserved as far as possible, i.e. if an index in the old region is also in the new region, then the array values with that index could be preserved in the arrays. It seemed initially that preserving the values was essential, but as the facilities were used in sample programs, it became clear that both schemes—destroy or preserve—were useful. Though preserve subsumes destroy—the preserved values can be overwritten—it is a waste to spend communication resources moving the data when it is not needed. So, ZPL has both a destructive assignment ($\langle==\rangle$) and a preserving assignment ($\langle=\#\rangle$) to support region change [41].

The ability to reallocate the data to the processors so conveniently has motivated new algorithms. For example, the standard solution for a 3D FFT specifies that a 1D FFT be performed along each of the three dimensions. The standard way to program this is to set up an allocation of the 3D array over a 2D arrangement of processors, keeping the dimension along which the 1D FFT is performed local to the processors. (This saves shipping data around in the complex butterfly pattern.) The array is then transposed to localize a different dimension, the 1D FFT is applied to that dimension, and another transpose is performed to localize the last dimension. The alternate way to program this in ZPL is to avoid the transposes and simply redistribute

the region across the processors. That is, the 3D region is mapped differently to the 2D processor grid. The details are complex, but result in better performance chiefly because the data is not repacked into the local portions of the array. That is, it saves local computation.

We continue to develop a better understanding of these facilities.

12. Problem Space Promotion

One aspect of ZPL not yet mentioned is that it has contributed to new parallel computation techniques, analogous to serial techniques like divide-and-conquer. One interesting example is Problem Space Promotion (PSP) [58]. **Problem Space Promotion** sets up a solution in which data of d dimensions is processed in a higher dimensionality that only logically exists in the same way the “tree” is only logical in the divide-and-conquer paradigm. The advantage of the PSP strategy is that it dramatically increases the amount of parallelism available to the compiler.

To explain PSP, consider the task of rank sorting n distinct values by making all n^2 comparisons, counting the number of elements less than each value, and then storing the item in the position specified by the count. (This is illustrative of an all-pairs type computation, and is not the recommended sorting algorithm, which is Sample Sort [58].) All-pairs computations, though common, would not normally be solved using n^2 intermediate storage for n inputs. Such a large intermediate array, even composed of one-bit values, would likely not fit in memory, and being lightly used has very poor cache performance.

Because of the flood operator and flood dimensions of ZPL, it is possible to perform this algorithm *logically* without actually creating the n^2 intermediate storage. The solution *promotes* the problem to a logical 2D space, by adopting the declarations

```
region R = [1, 1..n];           A "row" region
var V, S : [R] integer;         Input (V) and Output (S)
    FIR : [*,1..n] integer;     Flood array
    FIC : [1..n,*] integer;    Flood array
```

V and S are the input and output, respectively, and FIR —mnemonic for flood in rows—and FIC are helper arrays. They are logically 2D, but physically only contain (a portion of) the row or column.

The logic begins by initializing the two helper arrays using flood

```
[1..n,*] FIC := >>[1..n,1] V#[1, Index1];
```

where to set the FIC array, the input must first be transposed using remap (#) to change dimensions. If the input (in V) were 2 6 7 1 5, we would have these arrays:

$FIR:$	2 6 7 1 5	$FIC:$	2 2 2 2 2
	2 6 7 1 5		6 6 6 6 6
	2 6 7 1 5		7 7 7 7 7
	2 6 7 1 5		1 1 1 1 1
	2 6 7 1 5		5 5 5 5 5

The sorting—actually the construction of the permutation that specifies where the input must go to produce a sorted order—is a “one-liner”:

```
[R] S := +<<[1..n,1..n](FIC<=FIR);
```

which can be deconstructed as follows. The (logical) Boolean array showing which elements are smaller than another element,

```
FIC<=FIR:   1 1 1 0 1
             0 1 1 0 0
             0 0 1 0 0
             1 1 1 1 1
             0 1 1 0 1
```

is reduced to the vector to reorder the data by adding up the columns:

```
+<<[1..n,1..n](FIC<=FIR): 2 4 5 1 3
```

The columns are reduced because a comparison of the applicable region $R=[1..1..n]$ and the expression region in the partial reduction $[1..n,1..n]$ reveals that the first, i.e. columnar, dimension collapses.

A remap,

```
[R] S := v#[1,S];
```

produces the reordered data. A stable sort, i.e. a sort preserving the position of non-distinct input, only requires the addition of two subexpressions in the “one-liner”.

The key point about the PSP is that the processors are logically allocated in a 2D grid, so that the logical work is assigned to them all. With full parallelism and with the logically flooded arrays “banging” on the portion of the single row and column assigned to their processor, the performance is quite satisfactory [58] and the apparently impractical solution becomes practical for many n^2 and higher computations, including 3D matrix multiplication, n^3 n -body solutions, Fock matrix computations and more.

13. ZPL And HPF Comparisons

ZPL and HPF shared the same goal: Provide a global-view programming language for parallel computers and rely on the compiler to generate the right (communication, synchronization, etc.) code.

From that common goal the two efforts diverged, ZPL creating a new language and HPF parallelizing Fortran 90. Because these are such dissimilar approaches, there is little basis for comparison. The features in ZPL are the result of a from-first-principles design responding to the dictates of the CTA type architecture. The features in HPF came with Fortran 90, which is based on the RAM type architecture. Though we have observed, for example, that regions are better than array slices for data-parallel computation, the observation gives little insight about the two efforts because slices were given; HPF researchers studied different issues.

Though attempting such comparisons is not productive, I can say what benefits I think we got by our approach. Our type architecture approach and the CTA specifically gave the ZPL language designers and compiler writers the key parameters of the computers they were targeting: scalable P , unit time local memory reference, $\lambda \gg 1$ nonlocal memory references, etc. We could formulate programming abstractions to accommodate those properties. As compiler writers we could target the generic CTA, confident that the object code would run well (with comparably good performance) on any MIMD parallel computer. Further, ZPL programmers could analyze their code using the WYSIWYG model as a means of predicting program per-

formance and then observe that performance in fact. None of this was available to HPF.

One curious similarity is that neither language had a significant installed base of programs. ZPL was new, but in 1992 so was Fortran 90. In the early years it was difficult to find benchmark HPF programs to compare against.

Of course, it was not an advantage to the ZPL effort that most of the research community, nearly all of the funding, and most of the obvious corporate players were actively directed towards a different approach. The “group think” was pervasive. One difficult problem, common in reviews of papers and proposals, was for ZPL work to be dismissed because it didn’t address problems faced by HPF, that is, parallelizing sequential programs. A typical remark was “Of course you get outstanding performance; you changed the language to make it easy.” The result was that many referee reports did not speak to the merits (or lack thereof) of our papers.

Incidentally, the most foolish of the new-language criticisms was, “Design of new programming languages, including parallel programming, ceased to be even slightly interesting many, many years ago (circa 1980).” This was so narrow-minded (and wrong as Java, C#, Perl, Python, etc. prove) that the phrase “ceased to be interesting circa 1980” became a joke put-down among the team members. It always got a laugh.

We knew that ZPL was on the right track based on the performance numbers we were getting, but with so much of the community invested in HPF, ZPL’s success wasn’t welcome news. One paper—“ZPL vs HPF: A Comparison of Performance and Programming Style” [59]—demonstrated experimentally in 1995 that the ZPL approach was more effective than HPF. But the paper was not accepted *for partisan reasons*, as the conference program committee chairman admitted to me, and has never been published. It is interesting to speculate how a public debate in 1995 on the merits of the two approaches might have changed subsequent history.

The most serious consequence for ZPL of so much of the field being focused on one approach occurred towards the end of the 1990s, when we were extending A-ZPL and pushing on its flexibility: Essentially all funding for parallel programming research dried up, making it extremely difficult for us to complete our work. Speculating as to why this might have occurred, we can acknowledge unrelated events such as changes at funding agencies, the distraction of the “dot com boom,” etc. But, based on discussions at the time, the most likely explanation was the dawning realization by many in the field—researchers, funders, users—as to what the expenditure of so much talent and treasure on HPF was likely to produce. Parallel programming quickly became *technologia non grata*.

Despite the difficulties there is today, for a “David-size” fraction of the people and money, a publicly available open source ZPL compiler, which seems to be used for classroom and research purposes. Further, ZPL concepts have significantly influenced the next generation of parallel languages, Cray’s Chapel [60] and IBM’s X10 [61].

14. Assessment

The ZPL Project started with three goals: performance, portability and convenience, and it is essential to ask: how did we do? Though readers can consult the literature for detailed evaluations and judge for themselves, I am ex-

tremely pleased. As evidence consider data from a detailed analysis of the NAS MG benchmark [62, 63]. Our paper focuses on languages with implemented compilers as of 2000 with published results for the MG benchmark over the standard range of datasets. It contains evidence supporting all three goals.

Figure 13 gave the convenience data from the paper, as measured by lines of code. ZPL is nearly the most succinct of the programs. As explained earlier, this version of ZPL did not have first-class regions or directions, resulting in code less elegant than it should be; with the revisions to hierarchical arrays [33], however, the program would be

more readable, and perhaps slightly shorter. The key point to notice is that the computation portion of the Fortran 90 + MPI and CAF programs are not only much longer, but that the main constituent of the difference is communication code that is extremely difficult to write and debug. So, although these programs can give good performance, only ZPL, HPF and SAC truly simplify the programmer's task. (SAC is a functional language targeting only shared-memory machines.)

Figure 14 reproduces two pages of performance graphs from the analysis of the MG benchmark programs [63]. These tables reveal data about the performance the pro-

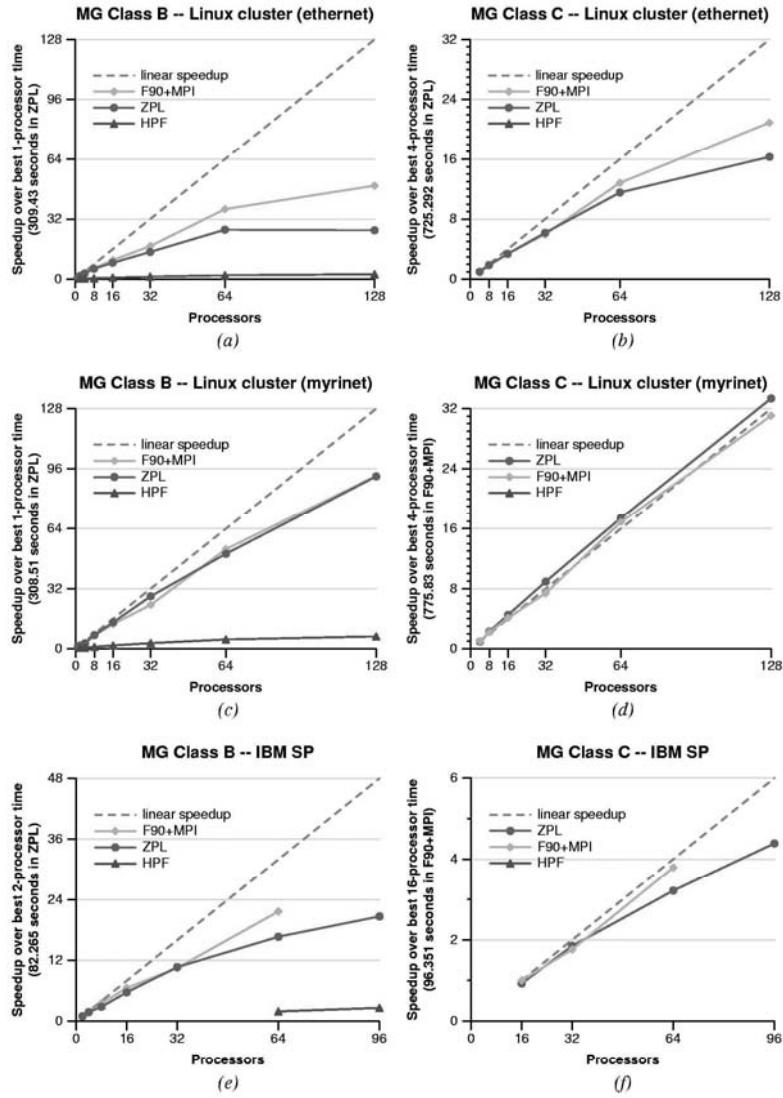


Figure 4: Speedup curves for classes B and C of MG on the Linux cluster using both ethernet and myrinet, and on the IBM SP. Note that there is not enough memory to run these problem sizes on small processor sets. Thus, we compute speedups for each graph using the fastest execution time on the smallest number of processors (indicated in the y-axis label). Due to its excessive memory allocation, the HPF version is unable to run on most configurations. Note that the F90+MPI version cannot run on 96 processors since it is not a power of two.

Figure 14. Performance Graphs from “A comparative study of the NAS MG benchmark across parallel languages and architectures” [63]. Speedup is shown; the dashed line is linear speedup.

grams can achieve across a series of machines, i.e. their portability.

Regarding portability, the figures show data for six different MIMD machines: two shared memory machines (Enterprise, SGI Origin), two distributed memory machines (Cray T3E, IBM SP-2), and two Linux clusters (Myrinet, Ethernet). These computers represent all the major architectures at the time except the Tera MTA [64], for which no MG program was available. (Performance numbers are missing in many cases, usually because the language does not target the machine or, for some HPF cases, because there was not enough memory.)

The performance numbers show that ZPL produces code that is competitive with the handwritten message passing code of F90 + MPI; further experiments and more detailed

analyses are found in the Chamberlain [33], Lewis [47] and Deitz [55] dissertations. Not only is the performance portable across the machines, ZPL scales well, generally doing better on larger problems. This is gratifying considering the programming effort required to write message-passing code compared to ZPL. It seems that for this sampling of computers, ZPL is comparable to the hand-coded benchmark, which was the performance-with-portability goal.

In closing, notice that the results are not given for the F90+MPI message passing program for processor values P that are not a power of 2. This is due to the fact that the programmers had to set up the problem by hand without the benefit of any parallel abstractions, and in doing so, they embedded the processor count in the program; making it a power of 2 was a simplifying assumption. In ZPL P is

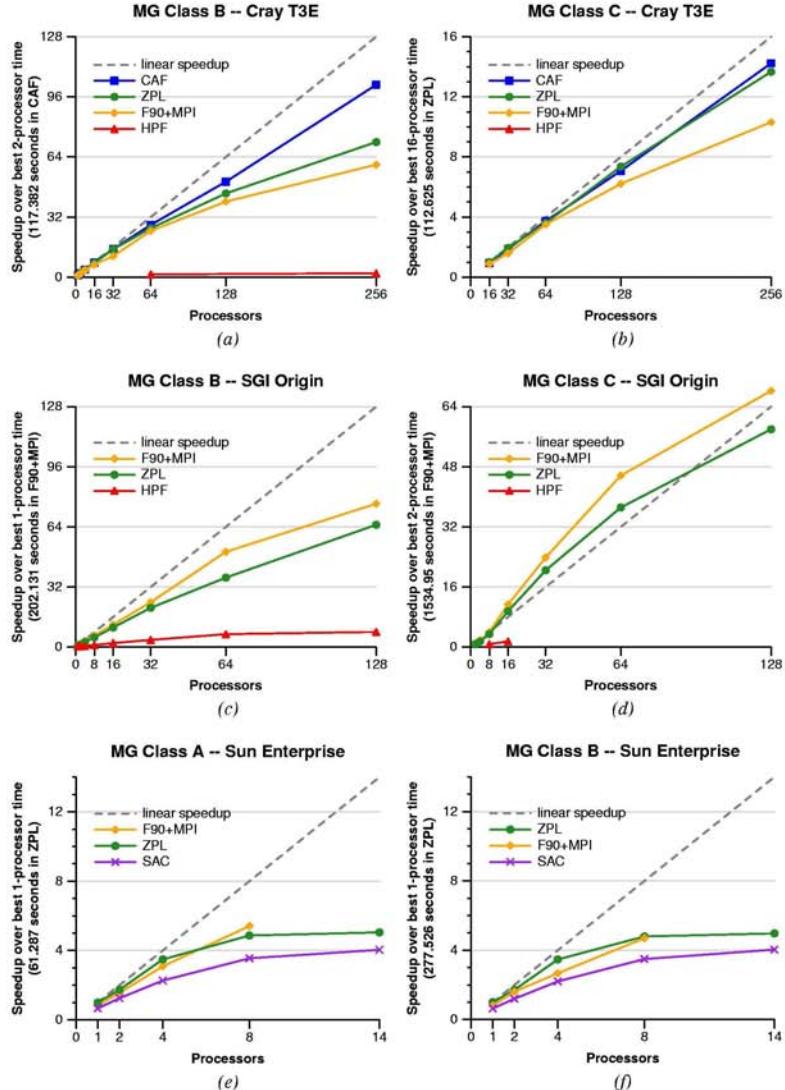


Figure 5: Speedup graphs for MG classes B and C on the Cray T3E and SGI Origin, and for classes A and B on the Sun Enterprise 5500. As in the previous figure, speedups for each graph are computed using the fastest execution time on the smallest number of processors. The superlinear speedup on the Origin is due to the memory traffic required to run a class C problem on 2 processors. We were unable to obtain reasonable timings on more than 128 Origin processors due to the network traffic involved in crossing between machines. See Appendix B for details.

Figure 14 (continued). **Performance Graphs** from “A comparative study of the NAS MG benchmark across parallel languages and architectures” [63].

assigned on the command line and the compiler sets up the problem anew for each run. This flexibility seems like a small thing. But it is symptomatic of a deeper problem: Building the parallel abstractions manually, as is necessary in message passing, is error prone. A year after the Figure 14 results were published, while running the ZPL Conjugant Gradient (CG) program on a 2000 processor cluster at Los Alamos, we discovered that for $P \geq 500$, the F90+MPI program didn't pass verification, though the ZPL did. What was wrong? The NAS researchers located the problem quickly; it was an error in the set-up code that was only revealed on very large processor configurations. This reminded us that one advantage to a compiler generating the code for an abstraction is that it only has to be made "right" once.

15. Summary

Though the ZPL project achieved its goals of performance, portability and convenience—making it the first parallel programming language to do so—the accomplishments of greatest long-term value may be the methodological results. The type architecture approach, used implicitly in sequential languages like Fortran, C and Pascal, has been applied explicitly to another family of computers for the first time. With silicon technology crossing the "multiple processors per chip" threshold at the dawn of the 21st century, there may be tremendous opportunities for architectural diversity; the type architecture methodology, we are beginning to see [19], allows a straight path to producing languages for new families of machines. Further, the WYSIWYG performance model transfers the compiler writer's know-how to the programmer in a way that has direct, visible and practical impact on program performance. As long as faster programs are better programs, models like WYSIWYG will be important.

Acknowledgments

It is with sincere thanks and with deepest appreciation that I acknowledge the contributions of my colleagues on the ZPL project: Ruth Anderson, Bradford Chamberlain, Sung-Eun Choi, Steven Deitz, Marios Dikaiakos, George Forman, Maria Gulickson, E Chris Lewis, Calvin Lin, Douglas Low, Ton Ngo, Jason Secosky, and Derrick Weathersby. There could not have been a research group with more energy, more smarts, more creativity or a higher "giggle index" than this ZPL team. Other grad student contributors included Kevin Gates, Jing-ling Lee, Taylor van Vleet and Wayne Wong. Judy Watson, project administrator, was a steady contributor to ZPL without ever writing a single line of code. There is a very long list of others to whom I'm also indebted, including grad students on my other parallel computation projects, faculty colleagues, research colleagues at other institutions, and scientists both at UW and elsewhere in the world. It has been a great pleasure to work with you all. Finally, for this paper I must again thank Calvin, Brad, E and Sung for their help, thank the tireless HoPL3 editors, Brent Hailpern and Barbara Ryder, for their patience, and the dedicated anonymous referees, who have so generously contributed to improving this paper.

References

- [1] Lawrence Snyder. Type architecture, shared memory and the corollary of modest potential. *Annual Review of Computer Science*, 1:289-317, 1986.
- [2] Gil Lerman and Larry Rudolph. *Parallel Evolution of Parallel Processors*, Plenum Press, 1993.
- [3] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon and D. W. Walker. *Solving Problems on Concurrent Processors*, Prentice-Hall, 1988.
- [4] Lawrence Snyder, The Blue CHiP Project Description. Department of Computer Science Technical Report, Purdue University, 1980.
- [5] Calvin Lin and Lawrence Snyder. ZPL: An array sublanguage. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*, 1993.
- [6] Calvin Lin and Lawrence Snyder. SIMPLE performance results in ZPL. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*, 1994.
- [7] Lawrence Snyder. Parallel programming and the poker programming environment. *Computer*, 17(7):27-36, July 1984.
- [8] Lawrence Snyder. Introduction to the configurable, highly parallel computer. *Computer*, 15(1):47-56, January 1982.
- [9] Richard O'Keefe, *The Craft of Prolog*, MIT Press, 1994.
- [10] Simon Peyton Jones and P.L. Wadler. A static semantics for Haskell. University of Glasgow, 1992.
- [11] Jorg Keller, Christoph W. Kessler and Jesper Larsson Traff. *Practical PRAM Programming*, John Wiley, 2000.
- [12] Richard J. Anderson and Lawrence Snyder. A comparison of shared and nonshared memory models of parallel computation. *Proceedings of the IEEE*, 79(4):480-487, April 1991.
- [13] Alan M.Turing. On computable numbers, with an application to the Entscheidungsproblem, Proc. London Math. Soc., 2(42):230-265, 1936.
- [14] John Backus. Can programming be liberated from the von Neumann style? *Communications of the ACM* 21(8):613-641, 1978.
- [15] J. T. Schwartz. Ultracomputers. *ACM Transactions on Programming Languages and Systems*, 2(4):484-521, 1980.
- [16] I.D. Scherson and A.S. Youssef. *Interconnection Networks for High-Performance Parallel Computers*. IEEE Computer Society Press, 1994
- [17] K. Bolding, M. L. Fulgham and L. Snyder. The case for Chaotic adaptive routing. *IEEE Trans. Computers* 46(12): 1281-1291, 1997.
- [18] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: towards a realistic model of parallel computation, *ACM Symposium on Principles and Practice of Parallel Programming*, 1993.
- [19] Benjamin Ylvisaker, Brian Van Essen and Carl Ebeling. A Type Architecture for Hybrid Micro-Parallel Computers. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2006.
- [20] David G. Socha. *Supporting Fine-Grain Computation on Distributed memory Parallel Computers*. PhD Dissertation, University of Washington, 1991.
- [21] Calvin Lin, Jin-ling Lee and Lawrence Snyder. Programming SIM-PLE for parallel portability, In U. Banerjee, D. Gelernter, A. Nicolau and D. Padua (eds.), *Languages and Compilers for Parallel Computing*, Springer-Verlag, pp. 84-98, 1992.
- [22] Calvin Lin. *The Portability of Parallel Programs Across MIMD Computers*. PhD Dissertation, University of Washington, 1992.
- [23] Ton Ahn Ngo and Lawrence Snyder. On the influence of programming models on shared memory computer performance. In *Proceed-*

- ings of the Scalable High Performance Computing Conference*, 1992.
- [24] Ton Anh Ngo. *The Role of Performance Models in Parallel Programming and Languages*. PhD Dissertation, University of Washington, 1997.
- [25] Raymond Greenlaw and Lawrence Snyder. Achieving speedups for APL on an SIMD distributed memory machine. *International Journal of Parallel Programming*, 19(2):111–127, April 1990.
- [26] Walter S. Brainerd, Charles H. Goldberg and Jeanne C. Adams. *Programmer’s Guide to Fortran 90*, 3rd Ed. Springer, 1996.
- [27] High Performance Fortran Language Specification Version 1.0 (1992) High Performance Fortran Forum, May 3, 1993. [34] Bradford L. Chamberlain. *The Design and Implementation of a Region-Based Parallel Language*. PhD Dissertation, University of Washington, 2001.
- [28] Bradford L. Chamberlain, Sung-Eun Choi, and Lawrence Snyder. A compiler abstraction for machine independent parallel communication generation. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*, 1997.
- [29] M. R. Haghigiat and C. D. Polychronopoulos. Symbolic analysis for parallelizing compilers. *ACM Transactions on Programming Languages and Systems*, 18(4):477–518, 1996.
- [30] J. R. Rose and G. L. Steele Jr. C*: An extended C language for data parallel programming. In *Proceedings Second International Conference on Supercomputing*, 1987.
- [31] Bradford L. Chamberlain, Sung-Eun Choi, E Christopher Lewis, Calvin Lin, Lawrence Snyder, and W. Derrick Weathersby. ZPL’s WYSIWYG performance model. In *Proceedings of the IEEE Workshop on High-Level Parallel Programming Models and Supportive Environments*, 1998.
- [32] Sung-Eun Choi. *Machine Independent Communication Optimization*. PhD Dissertation, University of Washington, March 1999.
- [33] Bradford L. Chamberlain. *The Design and Implementation of a Region-based Parallel Programming Language*. PhD Dissertation, University of Washington, 2001.
- [34] Bradford L. Chamberlain, E Christopher Lewis, and Lawrence Snyder. Problem space promotion and its evaluation as a technique for efficient parallel computation. In *Proceedings of the ACM International Conference on Supercomputing*, 1999.
- [35] Robert van de Geijn and Jerrell Watts. SUMMA: Scalable Universal Matrix Multiplication Algorithm. *Concurrency: Practice and Experience*, 1998.
- [36] R.E. Cypher, J.L.C. Sanz and L. Snyder. Algorithms for image component labeling on SIMD mesh connected computers. *IEEE Transactions on Computers*, 39(2):276-281, 1990.
- [37] MasPar Programming Language (ANSI C-compatible MPL) Reference Manual Document Part Number: 9302-0001 Revision: A3 July 1992.
- [38] Gregory R. Watson. *The Design and Implementation of a Parallel Relative Debugger*. PhD Dissertation, Monash University, 2000.
- [39] W. P. Crowley et al. The SIMPLE code. Technical Report UCID 17715, Lawrence Livermore Laboratory, February 1978.
- [40] Lawrence Snyder. *A Programmer’s Guide to ZPL*. MIT Press, Cambridge, MA, 1999. (The language changed in small ways and has been extended; it is now most accurately described in Chapter 2 of Chamberlain [33].)
- [41] Steven J. Deitz, Bradford L. Chamberlain, Sung-Eun Choi, and Lawrence Snyder. The design and implementation of a parallel array operator for the arbitrary remapping of data. In *Proceedings of the ACM Conference on Principles and Practice of Parallel Programming*, 2003.
- [42] Adam Beguelin, Jack Dongara, Al Geist, Robert Manchek , and Vaidy Sunderam. User guide to PVM. Oak Ridge National Laboratory, Oak Ridge TN 378 316367, 1993.
- [43] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI: the Complete Reference*. MIT Press, Cambridge, MA, USA, 1996.
- [44] Bradford L. Chamberlain, Sung-Eun Choi, and Lawrence Snyder. A compiler abstraction for machine-independent parallel communication generation. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*, 1997.
- [45] Sung-Eun Choi. *Machine Independent Communication Optimization*. PhD Dissertation, University of Washington, March 1999.
- [46] E Christopher Lewis, Calvin Lin, and Lawrence Snyder. The implementation and evaluation of fusion and contraction in array languages. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 1998.
- [47] E Christopher Lewis. *Achieving Robust Performance in Parallel Programming Languages*. PhD Dissertation, University of Washington, February 2001.
- [48] W. Derrick Weathersby. *Machine-Independent Compiler Optimizations for Collective Communication*. PhD Dissertation, University of Washington, August 1999.
- [49] Steven J. Deitz, Bradford L. Chamberlain, and Lawrence Snyder. Eliminating redundancies in sum-of-product array computations. In *Proceedings of the ACM International Conference on Supercomputing*, 2001.
- [50] Bradford L. Chamberlain, Sung-Eun Choi, E Christopher Lewis, Calvin Lin, Lawrence Snyder, and W. Derrick Weathersby. Factorjoin: A unique approach to compiling array languages for parallel machines. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*, 1996.
- [51] Philip S. Abrams. *An APL Machine*, PhD Dissertation. Stanford University, SLAC Report 114, 1970.
- [52] Ulrich Ruede. Mathematical and computational techniques for multilevel adaptive methods, *Frontiers in Applied Mathematics*, 13, SIAM, 1993.
- [53] Guy E. Bleloch. *Vector Models for Data-Parallel Computing*. MIT Press, 1990.
- [54] R. E. Ladner and M. J. Fischer. Parallel prefix computation. In *Proceedings of the IEEE International Conference on Parallel Processing*, 1977.
- [55] Steven J. Deitz. *High-Level Programming Language Abstractions for Advanced and Dynamic Parallel Computations*. PhD Dissertation, University of Washington, February 2005.
- [56] Steven J. Deitz, Bradford L. Chamberlain, and Lawrence Snyder. High-level language support for user-defined reductions. *Journal of Supercomputing*, 23(1), 2002.
- [57] E Christopher Lewis and Lawrence Snyder. Pipelining waveform computations: Experiences and performance. In *Proceedings of the IEEE Workshop on High-Level Parallel Programming Models and Supportive Environments*, May 2000.
- [58] Bradford L. Chamberlain, E Christopher Lewis, and Lawrence Snyder. Problem space promotion and its evaluation as a technique for efficient parallel computation. In *Proceedings of the ACM International Conference on Supercomputing*, 1999.
- [59] Calvin Lin, Lawrence Snyder, Ruth Anderson, Brad Chamberlain, Sung-Eun Choi, George Forman, E. Christopher Lewis, and W. Derrick Weathersby. ZPL vs. HPF: A Comparison of Performance and Programming Style, TR # 95-11-05 (available online from the University of Washington CSE technical report archive).
- [60] David Callahan, Bradford L. Chamberlain, and Hans P. Zima. The Cascade High Productivity Language. In *9th International Workshop*

- on High-Level Parallel Programming Models and Supportive Environments* (HIPS 2004), pp 52-60, April 2004.
- [61] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *20th OOPSLA*, pp. 519-538, 2005.
 - [62] David Bailey, Tim Harris, William Saphir, Rob van der Wijngaart, AlexWoo, and Maurice Yarrow. The NAS parallel benchmarks 2.0. Technical Report NAS-95-020, Nasa Ames Research Center, Moffet Field, CA, December 1995.
 - [63] Bradford L. Chamberlain, Steven J. Deitz, and Lawrence Snyder. A comparative study of the NAS MG benchmark across parallel languages and architectures. In *Proceedings of the ACM Conference on Supercomputing*, 2000.
 - [64] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera Computer System. *ACM SIGARCH Computer Architecture News*, 18(3):1 - 6, 1990.
 - [65] Michael A. Hiltzik. *Dealers in Lightning: Xerox PARC and the Dawn of the Computer Age*, Harper Collins, 1999.
 - [66] Eric Steven Raymond and Rob W. Landley. *The Art of Unix Usability*, Creative Commons, 2004. <http://www.catb.org/~esr/writings/taouu/html/ch02.html>.
 - [67] Calvin Lin and Lawrence Snyder. A comparison of programming models for shared memory multiprocessors. In *Proceedings of the IEEE International Conference on Parallel Processing*, 1990.
 - [68] M. Kandemir, P. Banerjee, A. Choudhary, J. Ramanujam and N. Shenoy. A global communication optimization technique based on data-flow analysis and linear algebra. *ACM Transactions on Programming Languages and Systems* 21(6):1251-1297, 1999.
 - [69] A. J. Wagner, L. Giraud and C. E. Scott. Simulation of a cusped bubble rising in a viscoelastic fluid with a new numerical method, *Computer Physics Communications*, 129(3):227-232, 2000.
 - [70] Robert W. Numrich, John Reid, and Kieun Kim. Writing a multigrid solver using Co-array Fortran. In *Proceedings of the Fourth International Workshop on Applied Parallel Computing*, 1998.
 - [71] S. B. Scholz. A case study: Effects of WITH-loop-folding on the NAS benchmark MG in SAC. In *Proceedings of IFL '98*, Springer-Verlag, 1998.

Self

David Ungar

IBM Corporation
ungar@mac.com

Randall B. Smith

Sun Microsystems Laboratories
randall.smith@sun.com

Abstract

The years 1985 through 1995 saw the birth and development of the language Self, starting from its design by the authors at Xerox PARC, through first implementations by Ungar and his graduate students at Stanford University, and then with a larger team formed when the authors joined Sun Microsystems Laboratories in 1991. Self was designed to help programmers become more productive and creative by giving them a simple, pure, and powerful language, an implementation that combined ease of use with high performance, a user interface that off-loaded cognitive burden, and a programming environment that captured the malleability of a physical world of live objects. Accomplishing these goals required innovation in several areas: a simple yet powerful prototype-based object model for mainstream programming, many compilation techniques including customization, splitting, type prediction, polymorphic inline caches, adaptive optimization, and dynamic deoptimization, the application of cartoon animation to enhance the legibility of a dynamic graphical interface, an object-centered programming environment, and a user-interface construction framework that embodied a uniform use-mention distinction. Over the years, the project has published many papers and released four major versions of Self.

Although the Self project ended in 1995, its implementation, animation, user interface toolkit architecture, and even its prototype object model impact computer science today (2006). Java virtual machines for desktop and laptop computers have adopted Self's implementation techniques, many user interfaces incorporate cartoon animation, several popular systems have adopted similar interface frameworks, and the prototype object model can be found in some of today's languages, including JavaScript. Nevertheless, the vision we tried to capture in the unified whole has yet to be achieved.

Categories and Subject Descriptors: K.2 [History of Computing] Software – programming language design, programming environments, virtual machines; D.3.2 [Programming Languages] Object-Oriented Languages; D.3.3 [Programming Languages] Language Constructs and Features – data types and structures, polymorphism, inheritance; D.1.5 [Object-oriented Programming]; D.1.7 [Visual Programming]; D.2.6 [Programming Environments] Graphical environments, Integrated environments, Interactive environments; D2.2 [Design Tools and Techniques] User Interfaces, Evolutionary prototyping; D2.3 [Coding Tools and Techniques] Object-oriented programming; I.3.6 [Computing Methodologies] Computer Graphics – Interaction techniques

General Terms. Performance, Human Factors, Languages

Permission to make digital/hard copy of part of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date of appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Permission may be requested from the Publications Dept., ACM, Inc., 2 Penn Plaza, New York, NY 11201-0701, USA, fax:+1(212) 869-0481, permissions@acm.org

©2007 ACM 978-1-59593-766-7/2007/06-ART9 \$5.00

DOI 10.1145/1238844.1238853

<http://doi.acm.org/10.1145/1238844.1238853>

Keywords: dynamic language; object-oriented language; Self; Morphic; dynamic optimization; virtual machine; adaptive optimization; cartoon animation; programming environment; exploratory programming; history of programming languages; prototype-based programming language

1. Introduction

In 1986, Randall Smith and David Ungar at Xerox PARC began to design a pure object-oriented, dynamic programming language based on prototypes called Self [US87, SU95]. Inspired by Smith's Alternate Reality Kit [Smi87] and their years of working with Smalltalk [GR83], they wanted to improve upon Smalltalk by increasing both expressive power and simplicity, while obtaining a more concrete feel. A Self implementation team was formed, first by the addition of Ungar's graduate students at Stanford, and then by the addition of research staff when the group moved to Sun Labs in 1991. By 1995, Self had been through four major system releases.

Self's simplicity and uniformity, particularly in its use of message passing for all computation, meant that a new approach to virtual machine design would be required for reasonable performance. The Self group made several advances in VM technology that could be applied to many if not most object-oriented languages. The group also created an innovative programming environment that could host multiple distributed users, and pioneered novel graphical user interface techniques, many of which are only now seeing commercial adoption.

Although the present paper has just two authors, the Self project was a group effort. The other members' dedication, hard work and brilliance made Self what it is. Those people are: Ole Ageson, Lars Bak, Craig Chambers, Bay-Wei Chang, Urs Hözle, Elgin Lee, John Maloney, and Mario Wolczko. In addition, our experience was deeply enriched by Ole Lehrmann Madsen, who spent a year with us as a visiting professor. We also appreciate the efforts of Jecel Assumpao who, over the years, has maintained a web site and discussion list for Self. We are indebted to the institutions that supported and hosted the Self project: Sun Microsystems, Stanford University, and Xerox PARC. While at Stanford, the Self project was generously supported by the National Science Foundation Presidential Young Investigator Grant #CCR-8657631, and by IBM, Texas Instruments, NCR, Tandem Computers, and Apple Computer.

Work on the project officially ceased in June 1995, although the language can still be downloaded and used by anyone with the requisite computing environment. But the ideas in Self can readily be found elsewhere: ironically, the implementation techniques developed for Self thrive today in almost every desktop virtual machine for JavaTM, a language much more conservative in design. We feel deeply rewarded that some researchers have understood and even cherished the Self vision, and we dedicate this paper to them.

This paper has four general parts: history, a description of Self and its evolution, a summary of its impact, and a retrospective. We begin with our personal and professional histories before we met in 1986, and summarize the state of object-oriented lan-

guages at that time with special emphasis on Smalltalk, as it was an enormous influence. We also discuss the context at Xerox PARC during the period leading up to the design of Self, and describe Smith's Alternate Reality Kit, which served as inspiration for some of Self's key ideas. This is followed by a description of Self that emphasizes our thoughts at the time. Moving our viewpoint into the present, we assess the impact of the system and reflect upon what we might have done differently. Finally, we sum up our thoughts on Self and examine what has become of each of the participants (as of 2006).

2. Before Self

Nothing comes from nothing; to understand Self's roots, it helps to look at what PARC and Smith and Ungar were doing earlier.

2.1. Smith Before Self

Perhaps because his father was a liberal-minded minister, or perhaps because he was also the son of a teacher, Smith has always been fascinated by questions at the boundaries of human knowledge, such as "What is going on to make the universe like this?" Thus it was perhaps natural for him to enter the University of California at Davis as a physics major. But along the way, he discovered computers: he so looked forward to his first programming course that on the opening day he gave his instructor a completed program and begged him to enable student accounts so that this excited student could submit his deck of cards (which calculated the friction in yo-yo strings). Computing felt more open and creative than physics: unlike the physical universe, which is a particular way, the computer is a blank canvas upon which programmers write their *own* laws of physics. There really was no major in computing in those days; the computer was perceived as a big, expensive tool, and Smith happily stuck with his physics curriculum, getting his PhD at UCSD in 1981. He then returned to his undergraduate alma mater as a lecturer in the UC Davis Physics Department.

One of the mysteries of physics is that a few simple laws can explain a wide range of phenomena. Smith enjoyed teaching, and was always impressed that he could derive six months of basic physics lectures from $F=ma$. Much progress in physics seems to be about finding theories with increasing explanatory power that at the same time simplify the underlying model. The notion that simplicity equated to explanatory power would later manifest itself in his work designing Self.

The draw of computing inevitably won him over, and Smith stopped chasing tenure in Physics, taking his young family to Silicon Valley in 1983 so he could work at Atari Research Labs, then directed by Alan Kay. During that year a rather spectacular financial implosion took out much of Atari. Smith was one of only a few remaining research staff members when the company was sold in 1984 to interests who felt no need for research. Atari Labs were closed and Smith joined the Smalltalk group at Xerox PARC.

2.2. Ungar Before Self

When Ungar was about six and struggling to tighten a horse's saddle girth, his father would say "Think about the physics of it." What stuck was the significance of how one chose how to think about a problem. Sometime in his early teens, Ungar was inspired by the simultaneously paradoxical and logical power of Special Relativity. Still later, experience with APL in high school and college kindled his enthusiasm for dynamic languages. Then, as an undergraduate at Washington University, St. Louis, he designed a simple programming language.

In 1980, Ungar went to Berkeley to pursue a Ph.D in VLSI design. Eventually, he got a research assistantship working on VLSI design tools for Prof. John Ousterhout, and was also taking a class on the same topic. At that time, the only way for a Berkeley student to use Smalltalk was to make the hour-plus drive down to Xerox PARC. Dan Halbert, also in the VLSI class, was making that trip regularly (in Butler Lampson's car) to use Smalltalk for his doctoral research on programming by demonstration. Halbert gave a talk in the VLSI class on how well Smalltalk would support VLSI design by facilitating mixed-mode simulation. In a mixed-mode system, some blocks would be simulated at a high level, others at a low level, and Smalltalk's dynamic type system and message-passing semantics would make it easy to mix and match. This chain of events kindled Ungar's interest in Smalltalk.

Dan Halbert took Ungar down to PARC several times in late 1980 and demonstrated Smalltalk. After seeing Smalltalk's reactive graphical environment and powerful, dynamic language, Ungar was hooked. He yearned to solve real problems in Smalltalk without the long drive. He obtained an experimental Smalltalk interpreter, written at HP, but it ran too slowly on Berkeley's VAX 11/780. This frustration would completely change the focus of Ungar's dissertation work, redirecting him from VLSI to virtual machines (see section 2.4.4). In the summer of 1985, Ungar left Berkeley and began teaching at Stanford as an assistant professor. He completed his dissertation that academic year, and received his PhD in the spring of 1986.

2.3. Object-Oriented (and Other) Programming Languages Before Self

The design of Self was strongly influenced by what we knew of existing languages and systems. Here are a few languages that were in Ungar's mind as a result of his lectures at Stanford.

Simula was the first object-oriented language per se. In its first published description, Dahl and Nygaard stated that its most important new concept was quasi-parallel processing [DN66]. Its designers were trying to use computers to simulate systems with discrete events. A key insight was the realization that the same description could be used both for modeling and for simulation. They extended Algol 60 by adding "processes" (what would now be called coroutines) and an ordered set feature. A Simula process grouped related data and code together, and this grouping came to be thought of as object-oriented programming. Multiple instances of a process could be created, and "elements" were references to processes. Simula's designers felt it was important to keep the number of constructs small by unifying related concepts. Although Simula's influence on Self was profound, it was indirect: Simula famously inspired Alan Kay, who in the 1970s led the Smalltalk group at the Learning Research Laboratory in Xerox PARC.

Parnas [Parn72] explained key principles of object-oriented programming without ever using the word "object." He convincingly showed that invariants could be better isolated by grouping related code and data together, than by a pure subroutine-based factoring.

Hoare argued convincingly for simplicity in language design [Hoar73]. This paper was one of Ungar's favorites and influenced him to keep the Self language small. It is interesting in view of Self's lack of widespread adoption that this aesthetic can also be found in APL, LISP, and Smalltalk, but not in the very popular object-oriented programming languages C++ and Java.

C++ [Strou86] was created by Bjarne Stroustrup, who had studied with the Simula group in Scandinavia but had then joined the Unix group at Bell Laboratories. Stroustrup wanted to bring the benefits of object-orientation and data abstraction to a community accustomed to programming in C, a language frequently considered a high-level assembler. Consequently, C++ was designed as a superset of C, adding (among other things) classes, inheritance, and methods. (C++ nomenclature uses “derived class” for subclass and “virtual function” for method.) To avoid incompatibility with C at the source or linker levels, and to avoid adding overhead to programs that did not use the new features, C++ initially omitted garbage collection, generic types, exceptions, multiple inheritance, support for concurrency, and an integrated programming environment (some of these features made it into later versions of the language). As we designed and built Self in 1987, C++’s complicated and non-interpretive nature prevented us from being influenced by its language design. However, its efficiency and support for some object orientation led Ungar and the students to adopt it later as an implementation language and performance benchmark for Self; we built the Self virtual machine in C++, and aimed to have applications written in Self match the performance of those written in (optimized) C++.

APL [Iver79] was an interactive time-shared system that let its users write programs very quickly. Although not object-oriented, it exerted a strong influence on both Smalltalk and Self. Ingalls has reported its influence on Smalltalk [Inga81], and APL profoundly affected Ungar’s experience of computing. In 1969, Ungar had entered the Albert Einstein Senior High School in Kensington, Maryland, one of only three in the country with an experimental IBM/1130 time-sharing system. Every Friday afternoon, students were allowed to program it in APL, and this was Ungar’s first programming experience. Though Ungar didn’t know it at the time, APL differed from most of its contemporaries: it was dynamically typed in that any variable could hold a scalar, vector, or matrix of numbers or characters. APL’s built-in (and user-defined) functions were polymorphic over this range of types. It even had operators: higher-order functions that were parameterized by functions. The APL user experienced a live workspace of data and program and could try things out and get immediate feedback. Ungar sorely missed this combination of dynamic typing, polymorphism, and interpretive feel when he went on to learn such mainstream languages as FORTRAN and PL/I.

Ungar’s affection for APL led to a college experience that had a profound impact. As a freshman at Washington University, St. Louis, in 1972, Ungar was given an assignment to write an assembler and emulator for a simple, zero-address computer. The input was to consist of instructions such as:

```
push 1
push 2
add
```

The output was to be the state of the simulated machine after running the given assembly program. His classmates went upstairs and, in the keypunch room (which Ungar recalls as always baking in the St. Louis heat) began punching what eventually became thick card decks containing PL/I programs to be run on the school’s IBM System/360. His classmates built lexers, parsers, assemblers, and emulators in programs about 1000 lines long; many of his classmates could not complete their work in the time allowed.

Meanwhile, Ungar’s fascination with APL had led to an arrangement permitting him to use the Scientific Time Sharing Corporation’s APL system gratis after hours. He realized that

with a few syntactic transformations (such as inserting a colon after every label), the assembler program to be executed became a valid APL program. Reveling in APL’s expressiveness, he wrote each transformation as a single, concise line of code. Then he wrote one-line APL functions for each opcode to be simulated, such as:

```
∇ADD X
PUSH POP + X
∇
```

Finally came the line of APL that told the system to run the transformed input program as an APL program. The whole program only took 23 lines of APL! This seemed too easy, but Ungar was unwilling to put in the hours of painstaking work in the keypunch sweatbox, so he turned in his page of APL and hoped he would not flunk. When the professor rewarded this unorthodox approach with an A, Ungar learned a lesson about the power of dynamic languages that stayed with him forever.

In retrospect, any student could have done something similar in PL/I by using JCL (IBM System/360 Job Control Language) to transform the program to PL/I and then running it through the compiler. But none did, perhaps because PL/I’s non-interpretive nature blinkered its users. Ungar always missed the productivity of APL and was drawn to Smalltalk not only for its conceptual elegance, but also because it was the only other language he knew that let him build working programs as quickly as in the good old days of APL. The design of Self was also influenced by APL; after all, APL had no such thing as classes: arrays were created either ab initio or by copying other arrays, just as objects are in Self.

2.4. Smalltalk

Smalltalk [Inga81] was the most immediate linguistic influence on Self. Smalltalk’s synthesis of language design, implementation technology, user interface innovation and programming environment produced a highly productive system for exploratory programming. Unlike some programming systems, Smalltalk had a principled design. Ingalls enumerated the principles in [Inga81], and many of them had made a strong impression on Ungar at UC Berkeley. We embraced these values as we worked on Self. Table 2 on page 39 enumerates these principles and compares their realizations in Smalltalk, the Alternate Reality Kit (described in section 2.6), and Self.

2.4.1. Smalltalk Language

It is truly humbling to read in HOPL II about Alan Kay’s approach to the invention of Smalltalk [Kay93]. Starting from notions of computation that were miles away from objects, Kay tells of years of work that produced a pure object-oriented environment including an interactive, reactive user interface and programming environment. Smalltalk introduced the concept (and reality) of a world of interacting objects, and we sometimes feel that Self merely distilled Smalltalk to its essentials (although we hope that Self made contributions of its own).

Smalltalk-76 introduced the concept of a purely dynamically typed object-oriented language. A Smalltalk computation consists solely of objects sending messages to other objects. To use an object, one sends a *message* containing the name of the desired operation and zero or more arguments, which are also objects. The object finds a method whose name matches the message, runs the method’s code, and returns an object as a result. Thus, the process that the reader may know as “method

invocation” in Java is called “message sending” in Smalltalk. The “class” is central to this story: every object is an “instance” of some class and must have been created by that class. A window on the screen is an instance of class `Window`, 17 is an instance of class `Integer`, and so on. Classes are themselves objects, and classes are special in that they hold the methods with which possessed by each of its instances (the *instance* variables). Also, a class typically specifies a superclass, and objects created from the class also possess any variables and methods defined in the superclass, the super-superclass, etc. Thus, all objects belonging to a given class possess the same set of variables and methods. Variables are dynamically typed, in that any variable can refer to any object, but that object had better respond to all the messages sent to it, or there will be a runtime error. Methods are selected at run time based on the class of the receiver.

In addition to the two pseudo-variables “self,” denoting the current receiver, and “super,” denoting the current receiver but bypassing method lookup in its class, there are six kinds of genuine variables: global variables, pool variables which pertain to every instance of a class in a set of classes, class variables which pertain to every instance of and to a given class, instance variables which pertain to a single instance, temporary variables of a method, and arguments. An instance variable can be accessed only by a method invocation on its holder, while temporaries and arguments pertain only to the current method invocation. (Arguments differ from the other kinds of variables in being read-only.)

By the time we had started working with Smalltalk, it had evolved from Smalltalk-76 to Smalltalk-80. This new version cleaned up several aspects of the language but also introduced a complicating generalization that would later motivate us to eliminate classes entirely. In Smalltalk-72, classes were not objects, but, according to Dan Ingalls, as the Smalltalk group “experienced the liveliness” of that system, they realized it would be better to make classes be objects. So, in Smalltalk-76, all classes were objects and each class was an instance of class `Class`, including class `Class` itself. That meant each class had the same behavior, because class `Class` held the common behavior for all classes. In Smalltalk-80, each class was free to have its own behavior, a design decision that brought a certain utility and also seemed in keeping with the first-class representation of classes as objects. However, it also meant that a class had to be an instance of some unique class to hold that behavior. The class of the class was called the metaclass. Of course, if the metaclass were to have its own behavior, it would require a meta-metaclass to hold it, and thus Smalltalk-80 presented the programmer with a somewhat complex and potentially infinite world of objects that resulted from elaborating the “instance of” dimension in the language. Smalltalk-80 makes this meta-regress finite by using a loop structure at the top of the meta-hierarchy, but many users had a lot of trouble understanding this. Although this could be seen as a poor design decision in going from Smalltalk-76 to Smalltalk-80, it might be argued that this is a problem one is forced to confront whenever classes are fully promoted to object status. Either way, this conceptually infinite meta-regress and the bafflement it caused new Smalltalk-80 programmers gave us a strong push to eliminate classes when we designed `Self`. As we look back at Smalltalk-80 in 2006, it seems to us that, given the desire for a live and uniform system, the instance-class separation sprouted into a tangle of conceptually infinite metaclasses that would seem inevitable if an entity cannot contain its own description.

2.4.2. Smalltalk Programming Environment

In addition to learning the Smalltalk language, the user also had to master a programming environment that came with its own organizational concepts. The Smalltalk programming environment was astounding for its time—it introduced overlapping windows and pop-up menus, for example—and exerted a strong influence on the `Self` project.

The programming environment used by Smalltalk programmers centers on the browser, inspector, and debugger. There are a few other tools (e.g., a method-by-method change management tool), but these three deliver much of what the programmer needs, and even these three share common sub-components. Hence, even in the Smalltalk programming environment, there was a sense of simplicity. Ironically, even though simple, the environment delivered features we miss when using some modern IDEs for languages such as Java. For example, one Java IDE in common use contains several times the number of menu items available in the Smalltalk tools, yet there is no way to browse a complete class hierarchy.

The “learnability” aspect of the Smalltalk programming environment was a key concern of the Smalltalk group when Smith joined it in 1984. The PARC Smalltalk group had descended from the Learning Research Group, which focused on the educational value of programming systems. Many in the group were aware that the Smalltalk-80 system was somewhat more difficult to pick up than they had hoped in the earlier days, and saw that the programming environment, being what the user sees, must have been largely responsible. Alan Kay had envisioned the Dynabook as a medium in which children could explore and create, and had conceived of Smalltalk as the language of the Dynabook. Hence one sensed a kind of subtext floating in the halls like a plaintive, small voice: “What about the children?” Although Smalltalk had started off as part of this vision, that vision had somehow become supplanted by another: creating the ultimate programmer’s toolkit.

The browser, the central tool for the Smalltalk programmer, was the result of years of enhancement and redesign. It is fair to say it does an excellent job of enabling users to write their Smalltalk code, and it has served as a model for many of today’s IDEs (though some bear a closer resemblance than others). The browsers feature small titled panes for selecting classes from within a category and methods within a class, plus a larger, central text pane for editing code. However, by the time it was released in Smalltalk-80, the browser had come to present a system view significantly removed from the underlying execution story of objects with references to one another, sending messages to each other. The standard Smalltalk-80 browser presents the user with notions such as categories (groups of classes), and protocols (bundles of methods), neither of which has a direct, first-class role in the Smalltalk runtime semantics of the program. For example, before a programmer can try creating even the simplest class, she must not only give the class a name, which may seem logical, but also decide on a System Category for the class, even though that category has nothing to do with the class’s behavior. Furthermore, the standard Smalltalk-80 browser features a prominent and important “instance/class” switch that selects either methods in the selected class or methods in the selected class’s class (the metaclass). Recall that a class, since it is an object, is itself an instance of some class, which would hold methods for how the class behaves, such as instantiation, access to variables shared amongst all instances, and the like. But what about the class’s class’s class? And the class’s class’s class’s class, and so on? One finds no extra switch positions for presenting those methods. Furthermore, if the pur-

pose of the browser is to show the methods in any class, why is the switch even needed?

At a deeper level, it was obvious to us that the use of tools, as great as they were, tended to pull one away from the sense of object. That is, the inspector on a Smalltalk hash table was clearly not itself the hash table. This was a natural outgrowth of the now famous Model View Controller (MVC) paradigm, invented by the PARC Smalltalk group as the framework for user interfaces. Under the MVC scheme, the view and controller were explicitly separate objects that let the user see and interact with the model. This was an elegant framework, but we questioned it. If the metaphor was direct manipulation of objects, then we thought that the UI and programming environment should give a sense that what one saw on looking at a hashtable actually *was* the hashtable. In section 2.6 on the Alternate Reality Kit and in sections 4.3 and 5.3 on user interface designs for Self, we discuss our approaches to providing a greater sense of direct object experience.

2.4.3. Smalltalk User Interface

To set the stage for the Smalltalk user interface, we first describe the state of user interface work when we met Smalltalk. Overlapping windows were first used in Smalltalk, and the early Smalltalk screens would look familiar even today. In those days at PARC and Stanford it was not uncommon to argue over a tiled-windows versus messy-desktop paradigm, though the latter ultimately came to dominate. HCI classes would discuss direct manipulation as though it were a somewhat novel concept, and everyday computer users were not clear whether the mouse-pointer window paradigm had real staying power, as it seemed to pander to the novice. In fact, the acronym Window Icon Mouse Pointer (WIMP) was often used derisively by those who preferred the glass teletype. Smith recalls that in some of his user studies it would take subjects roughly 30 minutes to get used to the mouse.

At the time Smalltalk was being designed, each application had its user interface hard-wired so that its implementation was inaccessible to the user; the interface could neither be dissected nor modified. Smalltalk was a breed apart: its user interface was itself just another Smalltalk program that ran in the same virtual machine as the programmer's own applications. Thus, by pointing the mouse at window W and hitting "control-C" to invoke the Smalltalk debugger, one could find oneself browsing the stack of an interrupted thread that handled UI tasks related to window W. One could then use this debugger to modify the code and resume execution to see the effects of the changes. Most of us hoped that something like that would eventually take over the world of desktop computing, but today that dream seems all but dead. There is no way to get into your word processor and modify it as it runs, though in those days, that would have been routine for the curious Smalltalk user.

At PARC in the early 1980s, researchers could sense how user interface innovations created down the hall were sweeping through the entire world. Silicon Valley researchers just assumed that the computer desktop UI was still fertile ground for innovation, feeling that the basic notions of direct manipulation would probably stick, so that invention would most fruitfully occur within that broad paradigm. We were smitten with direct manipulation and wanted to push it to an extreme. In particular, we were fascinated by the notion that the computer presents the user with a synthetic world of objects. It felt to us that the screens we saw in those days hosted flat, 2D, static pictures of objects. We wanted to feel that those were real objects, not pictures of them. This desire for "really direct manipulation"

consciously motivated much of our work and would show up first in the Alternate Reality Kit, as described in section 2.6, and ultimately in Self.

2.4.4. Implementation Technology for Smalltalk and Other Interpreted Languages

Although much work had been done to optimize LISP systems that ran on stock hardware, Ungar was not very aware of that work when the Self system was built. The contexts are so different and the problems differ enough that it is hard to say what would have been changed had he known more about LISP implementations. Ungar was familiar with the LISP machine [SS79], but as it was a special-purpose CISC machine for LISP, he felt it would not be relevant to efficient implementation of Self on a RISC.

In contrast, it is quite likely that Ungar, although not consciously aware of it at the time, was inspired by APL when he came up with the technique of customization for Self (section 4.1.1). As mentioned above, any variable in APL can hold a scalar, a vector or a matrix at any time, and the APL operations (such as addition) perform computation that is determined upon each invocation. For example, the APL expression $\mathbf{A} + \mathbf{B}$ executed three times in a loop could: add two scalars on its first evaluation, add a scalar to each element of a matrix on its second evaluation, and add two matrices element-by-element on its third. Although the computation done for a given operation could vary, the designers of the APL\3000 system [John79] observed that it was often the same as before. They exploited this constancy by using the runtime information to compile specialized code for expressions that would be reused if possible, thus saving execution time. If the data changed and invalidated code, it was thrown away and regenerated. Ungar had read about this technique years before implementing Self, and it probably inspired the idea that the system could use different compiled versions of the same source code, as long as the tricks remained invisible to the user.

When Smalltalk was developed in the early to mid 1970s, commercially available personal computers lacked the horsepower to run it. Smalltalk relied on microcode interpreters running on expensive, custom-built research machines. Developed in house at Xerox PARC, these machines (called Altos [Tha86], later supplanted by Dolphins, and then Dorados) were the precursors of 1990s personal computers. The Dorado was the gold standard: it was fast for its time (70ns cycle time), but had to be housed in a separate air-conditioned room; a long cable supplied video to the user's office. These expensive and exotic machines allowed the PARC researchers to live in a world of relatively abundant cycles, personal computers, and bitmapped displays years before the rest of us.

Even with this exotic hardware, Smalltalk's implementers at PARC had to resort to compromises that increased performance at the cost of flexibility. For example: arithmetic, identity comparison, and some control structures were compiled to dedicated bytecodes whose semantics were hard-wired into the virtual machine. Thus, the source-level definitions of these messages were ignored. A programmer, seeing the definitions, might think that these operations were malleable, edit the definition and accept it, yet nothing would change. For example, Smith once changed the definition of the if-then-else message to accept "maybe" as the result of comparisons involving infinity. He was surprised when, though the system displayed his new definition, it kept behaving in accordance with the old one. And Mario Wolczko, who taught Smalltalk before joining the Self group, once had a student create a subclass of Boolean, only to discover

that it did not work. The Self system was built later and enjoyed the luxury of more powerful hardware. Thus, it could exploit dynamic compilation to get performance without sacrificing this type of generality (section 5.1).

In 1981, Ungar built his own Smalltalk system, Berkeley Smalltalk (BS). Its first incarnation followed the “blue book” [GR83], which used 16-bit object pointers, an object table, and reference counting.¹ This kind of object pointer is known as an indirect pointer, because instead of pointing directly to the referenced object, it points to an object table entry that in turn points to the referenced object. This indirection doubles the number of memory accesses required to follow the pointer and therefore slows the system. L. Peter Deutsch, an expert on dynamic language implementations who had worked on the Dorado Smalltalk virtual machine [Deu83] at Xerox PARC, began a series of weekly tutoring sessions on Smalltalk virtual machines with Ungar. Deutsch had just returned from a visit to MIT, where he was probably inspired by David Moon to suggest that Ungar build a system that handled new objects differently from old ones. After obtaining promising results from trace-driven simulations, Ungar rewrote Berkeley Smalltalk to use a simple, two-generation collection algorithm that he called Generation Scavenging [Ung84]. Ungar realized that, in addition to directly increasing performance by reducing the time spent on reclamation, this collector would indirectly increase performance by making it possible to eliminate the object table. This optimization was possible because Generation Scavenging moved all the new objects in the same pass that found all pointers to new objects and could thus use forwarding pointers. In addition, since most objects were reclaimed when new, old objects were allocated so rarely that it was reasonable to stop the mutator for an old-space reclamation and compaction and thus again use forwarding pointers. The resulting system was the first Smalltalk virtual machine with 32-bit pointers and the first with generational garbage collection. Ungar told Deutsch about his excitement at removing the overhead of pointer indirection involved with the object table. Deutsch didn’t share the excitement; he estimated the speedup would be less than 1.7. Ungar disagreed, and talked Deutsch into betting a dinner on it. So, when the new algorithm was running, Ungar tuned and tuned till it was 1.73 times faster than the previous tuned version of Berkeley Smalltalk: Deutsch treated Ungar to a very fine dinner in a Berkeley restaurant. As of this writing (2006), almost all desktop- and server-based object-oriented virtual machines use direct pointers, thanks perhaps in part to Deutsch’s willingness to make a bet and graduate student Ungar’s desire to prove himself to Deutsch and claim a free meal.

At Berkeley, Deutsch and Ungar continued their discussions. When the Sun-1 came out, Deutsch decided to build a system based on dynamic compilation to native code and inline caching that would let him run Smalltalk at home [DS84]. Deutsch and Schiffman’s PS (“Peter’s Smalltalk”) system was in many ways the precursor of all dynamically compiling object-oriented virtual machines today. After Ungar spent a few months trying to optimize his interpreter and receiving only diminishing returns, he realized that only a compilation-based virtual machine (such as PS) could yield good performance. It was this experience that led Ungar to rely on compilation techniques for the Self system.

1. The “blue book” (our affectionate name for the first book on Smalltalk) was the authoritative guide (since it was the only one) and contained the code (in Smalltalk) for a reference implementation. Ungar recalls that Dave Robson used to call this Smalltalk-in-Smalltalk as “the slowest Smalltalk virtual machine in the world.”

Meanwhile, during the 1980-1981 academic year, Berkeley professor David Patterson was finishing up his Berkeley RISC project, demonstrating that a simple instruction-set architecture with register windows could run C programs very effectively. By eliminating the time spent to save and restore registers on most subroutine calls, the RISC architecture could execute the calls very quickly. Ungar and others at Berkeley saw a match between RISC’s strengths and the demands of a Smalltalk implementation. Patterson saw this too, and in collaboration with Prof. David Hodges started the Smalltalk on a RISC (SOAR) [PKB86, Ung87] project. Based on a simple RISC machine, SOAR added some features to support Smalltalk and relied on a simple ahead-of-time compiler [BSUH87] to attain 70ns-Dorado-level performance on a (simulated) 330ns microprocessor. The rack-sized Dorado ran at a clock speed of 14 MHz, while the (simulated) chip-sized SOAR microprocessor ran Smalltalk just as fast with a mere 3 MHz clock. (Today, in 2006, commercial microprocessors run at clock speeds about a thousand times faster than SOAR’s, and have no trouble at all with interpreted Smalltalk.) This system was another proof that compilation could hold the key to dynamic object-oriented performance.

For his doctoral research, Ungar helped design the instruction set, wrote the runtime system (in SOAR assembler) and ran benchmarks. Then he removed one architectural feature at a time and substituted a software solution so as to isolate the contribution of each feature. One of the most important lessons Ungar learned from the project was that almost all the system’s “clever” ideas had negligible benefit. In fact, the vast bulk of speed improvements accrued from only a few ideas, such as compilation and register windows. In his dissertation, he called the temptation to add ineffective complexity “The Architect’s Trap.” A few years later, in 1988 and 1989, Ungar had to relearn this lesson in the evolution of the Self language, as described in section 4.2.

In 1988, after PS had been completed and Ungar had graduated, the Smalltalk group at Xerox PARC spun off a startup company called ParcPlace Systems to commercialize Smalltalk. For their ObjectWorks product, they built a Smalltalk virtual machine called HPS. Extending the ideas in PS, HPS used Deutsch’s dynamic translation technique and a clever multiple-representation scheme for activation records. Unlike PS, it was written in C, not assembler, and employed a hybrid system for automatic storage reclamation. The latter, on which Ungar consulted, comprised a generation scavenger for new objects and an incremental, interruptible mark-sweep collector for the old objects. An object table permitted incremental compaction of the old objects. When it was built, around 1988, it was probably the fastest Smalltalk virtual machine, and its success with dynamic translation served as an inspiration.

2.5. Xerox PARC in the 1980s

By the early 1980s Xerox PARC had established itself as the inventor of much of the modern desktop computer. At a time when most of us in the outside world were just becoming comfortable with time-shared screen editors running on character-mapped displays that showed 25 lines of 80 fixed-width characters, each PARC engineer had his own personal networked computer with keyboard, mouse, and a bit-mapped display showing multiple windows with menus and icons. They authored WYSIWYG documents, sent them to laser printers, e-mailed them to each other, and stored them on file servers. All this has now, of course, become commonplace.

But another part of the vision held by many at PARC was slower to materialize in the outside world, and in many ways never did. That dream depicted users as masters of their own computers, able to modify applications in arbitrary ways and even to evolve them into entirely new software. This vision of “everyman as programmer” was part of Alan Kay’s story that motivated the work of the PARC Smalltalk group [Kay93]. This group looked to the idea of dynamic, object-oriented programming as the underlying mechanism that would make the elements of the computer most sensibly manifest. Kay’s group had a tradition of attending to the user interface (the Smalltalk group had introduced the idea of overlapping windows) and of focusing on education. Kay and his group felt that students should be creators in a powerful and flexible medium and that a dynamic object-oriented language was the key enabler. Kay’s group had developed several versions of the Smalltalk language: Smalltalk-72, Smalltalk-76, and finally Smalltalk-80.

Alan Kay left PARC in 1982 but his group carried on under the leadership of Adele Goldberg. It had hosted a few researchers who created more visual programming environments such as Pygmalion [FG93], Rehearsal World [Smi93], and ThingLab [BD81]. These environments were written in Smalltalk but were themselves essentially visual programming languages, with somewhat different semantics from Smalltalk itself. For example, ThingLab took the user’s graphically specified constraints to generate code that maintained those constraints. Perhaps because it was a graphical environment, or perhaps because of SketchPad’s inspiration [Suth63], ThingLab’s designer, Alan Borning, presented users with a set of “prototype” objects to copy and modify in their work. The copying of a prototype became recognized as being a little deeper than it might seem at first glance; it offered an alternative to instantiating a class that felt much more concrete. This distinction may not seem very compelling in a compile-first/run-later environment such as Java or C, but in Smalltalk, where one is always immersed in a sea of running objects even while writing new code, the advantages of working with concrete instances was more apparent. Perhaps from similar intuitions, others had been exploring the idea of adding “exemplars” to Smalltalk [LTP86], instances that accompany the class hierarchy and serve as tangible representatives of the classes.

When Smith joined PARC in 1984, he would add to this list of visual programming systems written in Smalltalk by creating the Alternate Reality Kit, or ARK. Like ThingLab and SketchPad, ARK would be a construction environment based on prototypes.

2.6. ARK, The Alternate Reality Kit

Smith had always loved teaching physics. When he was lecturing in the UC Davis Physics Department, he felt the students became somewhat disconnected from the material when he covered topics such as relativity and quantum mechanics, because few if any demonstrations were available to provide a tangible connection to relevant physical experience. When he left academia for Silicon Valley research life at Atari Corporate Research in 1983, Smith started to investigate how a simulation environment might provide a tangible experience for learning relativity by letting students see what the world would be like if the speed of light were, say, 5 mph. He hoped that someday students playing in such a simulated world would obtain such an automatic and intuitive understanding of relativity that they would laugh off mental puzzlers such as the twin paradox as a trivial misunderstanding. When he joined PARC, Smith began to think about generalizing on his previous work. Smith began to realize that changing the speed of light to 5 mph was just an instance of a more powerful idea: a simulation can provide a

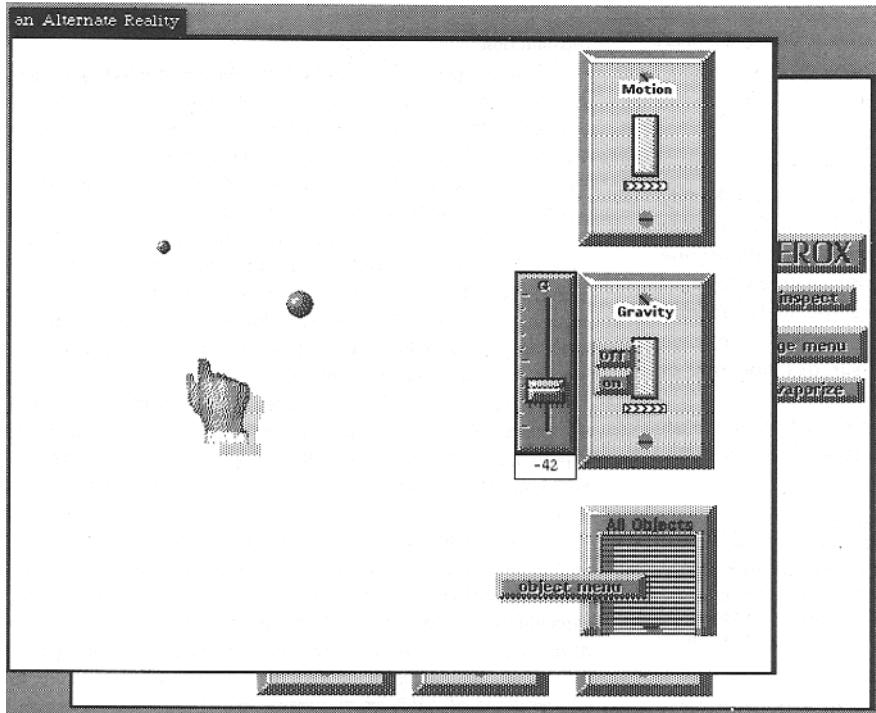
way for students to experience how the world is not, as well as how it is. In the real world, we are stuck with the laws of physics we have been given. In a simulation, we can see what role a law plays by watching what happens when we change it. Smith set to work to create an environment making it possible to create such simulations; because of this emphasis on changing the nature of reality, Smith called the system the Alternate Reality Kit. Smalltalk’s ability to change a program as it ran was the key to granting the ARK user the power to change physical law in an active universe.

The Alternate Reality Kit, implemented in Smalltalk-80, emerged as an open-feeling kit of parts, featuring lots of motion and subtly animated icons (see Figure 1). A user could grab objects, throw them around, and modify them in arbitrary ways through messages sent by buttons. For its time, the system had unusual, “realistically” rendered objects. The lighting model implied a third dimension, and most objects were intentionally drawn without an outline to remind the viewer of real-world objects, which also do not generally have outlines. A drop shadow for objects lifted “out of the plane” also provided a sense of a third dimension. Having only one-bit-deep displays meant all this had to be achieved with stipple patterns, requiring careful rendering and a little more screen real estate than might otherwise be required. This look would later be carried into the Self user interface. Today, drop shadows and pseudo-3D user interface elements with highlights and beveled edges are commonplace, and we are seeing more animation as well. ARK may have been the first system to include many of these ideas.

ARK also foreshadowed Self’s elimination of the class concept by sweeping Smalltalk’s classes under the rug. For example, it featured a “message menu” that the user could “pop up” directly on any display object and contained a list of every Smalltalk message to which the display object could respond. Selecting from the menu created a button that was attached to the object that could be pressed to send the message, then discarded if not needed, dropped onto other objects for use there, set aside, or simply left in place for future use. If the message required parameters, the button had retractable plugs that could be drawn out and dropped on the parameter objects. If the message returned a result, that object was made into a display object and popped up onto the screen. To create the menu of available messages, the underlying Smalltalk system started with the class of the display object and simply scanned up the class hierarchy, collecting the methods from each class as it went. As a result, the presence of a class was effectively hidden from the ARK user, even though classes were of course being used under the covers.

Furthermore, in ARK, any object could be modified and new kinds of state and behavior introduced within the simulation while everything was running. Unlike Smalltalk, ARK enabled the user to add an instance variable directly to an object, simultaneously specifying the name of the variable and its value. Because ARK was a Smalltalk program, making a new kind of object was implemented at the Smalltalk level as three steps: 1] make a new subclass specifying the new instance variable, 2] instantiate that class to make a new object O, and 3] replace the on-screen instance with O. In other words, the role of the Smalltalk class was again being hidden. The class was implementing something that in ARK felt not only more tangible but more to the point: working directly with instances.

Thus, even though ARK users worked directly with instances, they had full access to sending Smalltalk messages and making new kinds of objects. The notion of making a new kind of object simply by modifying an existing instance foreshadowed the pro-



tototype-based approach that was to be the basis of the Self object model. In ARK, it seemed unnecessary to even think about a class, and Self would have none.

A final aspect of ARK influenced the later design of Self (section 3). When a new instance variable was created for an object in ARK, it seemed natural to have the system automatically create “setter” and “getter” methods that would then show up in the message menu used for creating buttons. Thus the message menu presented a story based on the object’s behavior, hiding the underlying state. It was clear that with setter and getter methods, the full semantics of an object was available through message passing alone: any notion of state was hidden at a deeper implementation level.

A downside to automatically exposing instance variables through getters and setters is that it broadens the public interface to an object, and so might make it more difficult to change an object since other parts of a system might come to rely on the existence of these methods. Note that automatic getters and setters do not really violate the design principle of encapsulation, as the sender of a set or get message has no idea what kind of internal state (if any) is employed.

ARK also brought together some of the personalities who would later create Self. It was a demonstration of Smith’s ARK in late 1985 or early 1986 that made Ungar realize that he wanted to collaborate with Smith: Smith showed Ungar an ARK graphical simulation of load-balancing in which processes could migrate from CPU to CPU. Ungar suggested attaching a CPU to one process so that when the process migrated, it would take the (simulated) CPU with it. When Smith was able to do this by just sticking a CPU widget to a process, Ungar realized that there was something special here; ARK was the kind of system that appeared simple but let its users easily do the unanticipated. Ungar was so taken with ARK that he later used a video of it for the final exam in his Stanford graduate course on programming language design. When Bay-Wei Chang took this exam, he was inspired to join the Self project. The spirit that shone through ARK illuminated the path for Self.

Figure 1. The Alternate Reality Kit (ARK), an interactive simulation environment that was also a visual programming system. Some ideas in ARK influenced the design of Self. The screen shows several buttons, some attached to objects. New objects could be made by a copy-and-modify process, and any new state in the new object was accessed through new buttons. This foreshadowed Self’s use of prototypes and the way Self entirely encapsulates state behind a message-passing mechanism. ARK also had a feel of being a live world of moving, active objects that was unusual for its time and influenced the programming environment, as well as in some sense the deeper semantics and overall goals, of Self.

3. Self is Born at PARC

In 1985, as Smith was working on the Alternate Reality Kit, Ungar joined the faculty at Stanford. Stanford was just “down the hill” from PARC, and the Smalltalk group decided to bring Ungar in to collaborate with the group a few days per week. In 1981 the Smalltalk group had released Smalltalk-80 [GR83], the latest and perhaps most complete and commercially viable in the string of Smalltalk releases. The group considered it their natural charter to invent Smalltalk-next, and a follow-on to Smalltalk-80 was perhaps overdue. To tackle this design problem, the Smalltalk group decided to break into teams, each of which would propose a next language. Smith and Ungar paired up to create their own proposal for a language that would eventually become Self.

At the time we felt that Smalltalk was striving to realize a Platonic ideal, an apotheosis, of object-oriented programming. Smalltalk seemed to be heading toward a model in which computation proceeds by sending messages (containing objects as arguments) to objects and receiving objects in return. That’s all. There is nothing about bits. Once in a while, one of these messages might turn on a pixel on a display. But, really, the notion of computation rests on a higher plane than bits in memory and is more abstract. Ungar likened this model of computation to Rutherford’s experiments to learn about the atomic nucleus. Rutherford could not look inside an atom; he had to shoot subatomic particles at atoms and record how they bounced off. The pattern led him to deduce the existence of the nucleus. Similarly, we felt that there should be no way to look inside of an object; an object should be known only by its behavior, and that behavior could be measured only by the measurements on the behavior of objects returned in response to messages.

3.1. The Basic Ideas

When we started to design Self, we were partly inspired by ARK: we wanted the programming environment’s graphical display of an object to *be* the object for the programmer. We

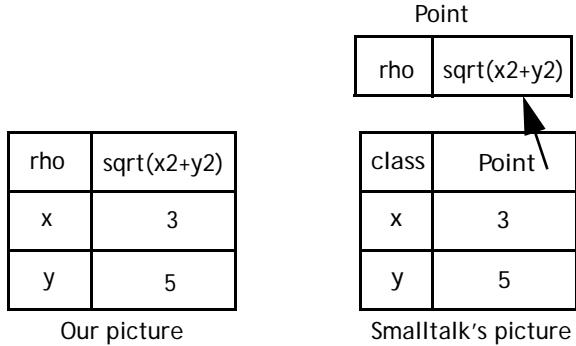


Figure 2. When we pictured a simple point object, we imagined it differently from in Smalltalk. In particular, the state and behavior of the object itself drew our attention, but the class did not. Since we wanted the language and environment level to mimic a hypothetical physical embodiment, we left classes out of Self. A Self object contains slots, such as rho and x in the figure, and a slot may function either as a holder of state (such as x) or as a holder of behavior (such as rho). (For simplicity of illustration, assume that the computed object is returned by the message send.)

noticed that whenever we drew an object on the whiteboard, our pictures were different from Smalltalk’s; our objects always looked like small tables (see Figure 2) with no classes in sight.

As mentioned above, we employed a minimalist strategy in designing Self, striving to distill the essence of object and message. A computation in Self consists solely of objects which in turn consists of slots. A slot has a name and a value. A slot name is always a string, but a slot value can be any Self object. A slot can be asterisked to show that it designates a *parent*.

Figure 3 illustrates a Self object representing a two-dimensional point with x and y slots, a parent slot called myParent, and two special assignment slots, x: and y:, that are used to assign to the x and y slots. The object’s parent has a single slot called print (containing a method object to print the point).

We found the resulting instance-oriented feel of the environment appealing because it lent more clarity and concreteness to a program design, with no loss of generality from Smalltalk-80. Additionally, Self’s design eliminated metaclasses, which were one of the hardest parts of Smalltalk for novices to understand,

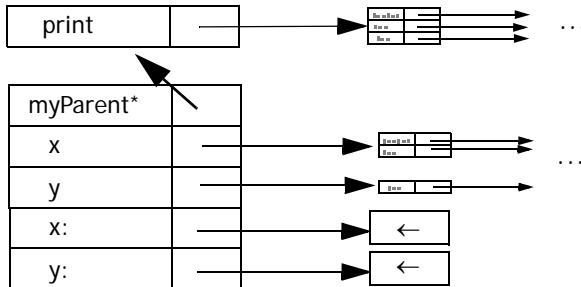


Figure 3. A Self point has x and y slots, with x: and y: slots containing the assignment primitive for changing x and y. The slot myParent carries a “parent” marker (shown as an asterisk). Parent slots are an inheritance link, indicating how message lookup continues beyond the object’s slots. For example, this point object will respond to a print message because it inherits a print slot from the parent.

and avoided Smalltalk’s long-standing schism between instance attributes and class attributes.² (The latter are also called “static methods and variables” in Java and C++.)

Recall that the novice Smalltalk-80 programmer had to learn about the scoping rules for each of Smalltalk’s six classes of variables (see Figure 4). Smith recalls thinking this was somehow an odd story for an object-oriented language, in which getting and setting state could be done with message passing to objects. Sometime soon after joining the Smalltalk group at PARC (possibly in 1982 or 1983), Smith mentioned this variable-vs.-message dichotomy to Dave Robson, a senior member of the Smalltalk group and co-author of the Smalltalk “blue book” [GR83]. Smith recalls Robson replying in a somewhat resigned tone, “Yeah, once you’re inside an object, it’s pretty much like Pascal.”

Ungar independently stumbled on the same question during a lunch in which Deutsch offhandedly suggested that these six types of variable accesses could be unified. We started to think of trying to use message sending as the only way to access storing and retrieval of state, and came up with a design that could merge all variable accesses with message passing (see Figure 5). We presented the design in an informal talk to the Smalltalk group in 1986, and in 1987 wrote the paper “Self: The Power of Simplicity” [US87].

We implemented inheritance with a variation on what Henry Lieberman called a “delegation” model [Lieb86]: when sending a message, if no slot name was matched within the receiving object, its parent’s slots were searched for an object with a matching slot, then slots in the parent’s parent, and so on. Thus our point object could respond to the messages x, y, x:, y:, and myParent, plus the message rho, because it *inherited* the rho slot from its parent. In Self, any object could potentially be a parent for any number of children and could be a child of any object. This uniform ability of any object to participate in any role of inheritance contributes to the consistency and malleability of Self and, we hope, to the programmer’s comfort, confidence, and satisfaction.

To accomplish this unification, we decided to represent computation by allowing a Self object optionally to include code in addition to slots. An object with code is called a *method*, since it does what methods in other languages do. For example, the object in the rho slot above includes code and thus serves as a method. However, in Self, any object can be seen as a method; we regard a “data” object (such as 17) as containing code that merely returns itself. This viewpoint unifies computation with data access: when an object is found in a slot as a result of a message send it gets *run*; a datum returns itself, while a method invokes its code. Thus, when the rho message is sent to our point object, the code in the object in the rho slot is found and that object’s method runs. This unification reinforces the interpretation that it is the experience of the client that matters, not the inner details of the object used to create that experience.

Self’s unification of variable access and message passing relied on the fact that a method would run whenever it was referenced.

2. Later, to support a programming environment, mirrors were added to Self. A mirror on an object contains information about that object, and may seem somewhat like a class that contains information about its instances. However, as discussed in section 4.4, an object may exist with no mirrors, unlike instances, classes, and metaclasses. Furthermore, had we been willing to guarantee that every object would transitively inherit from a root, we could have put reflective functionality in that root with no need for mirrors.

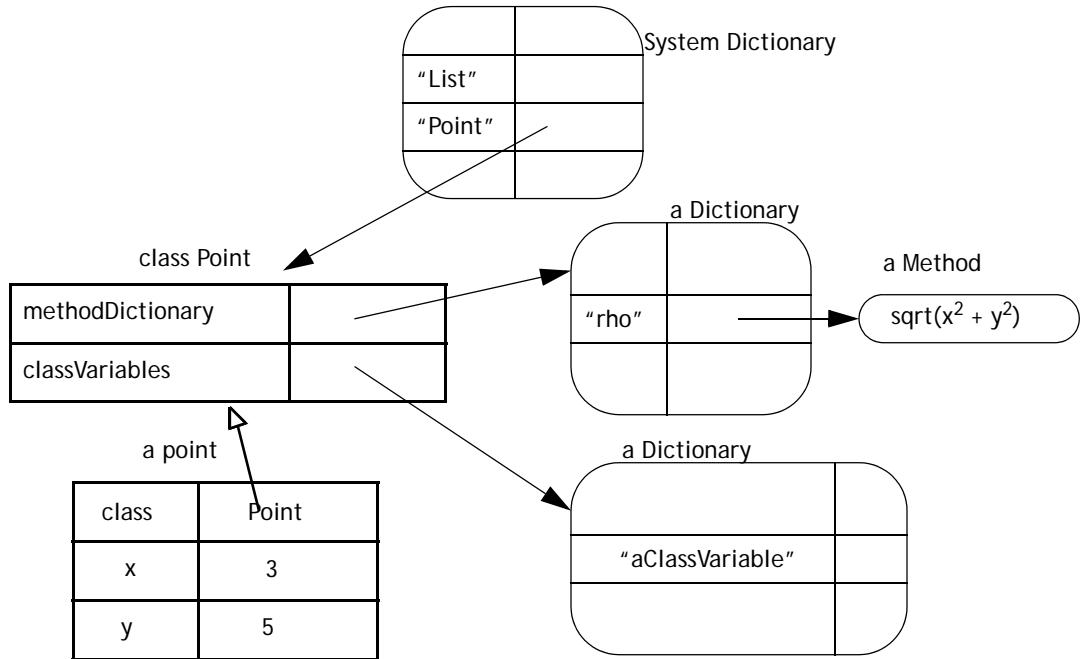


Figure 4. Smalltalk uses dictionary objects to hold the variables accessible from different scopes, though instance variables such as x and y for a point are directly available within the object. Class variables and global variables such as Point and List are held in such special dictionary objects with string objects (in quotes) as keys. Methods are also held in a special dictionary. All these dictionaries must exist in this way, as the entire language semantics relies on their existence. Not shown here are "Pool" variables, temporaries and arguments within a method context, or temporaries within a block closure object. In Figure 5 we show how Self achieves this scoping using objects and inheritance.

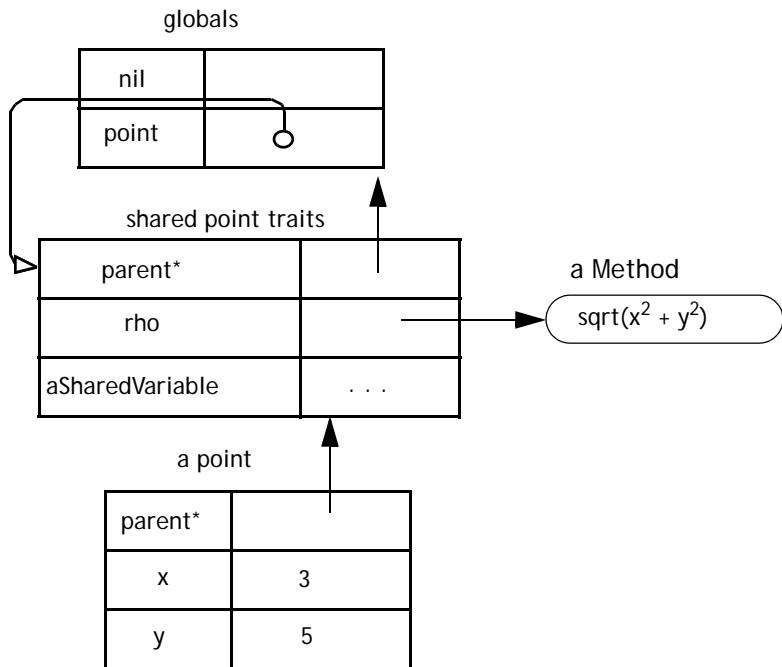


Figure 5. Self's object design gets many different scopes of variables for free. In Self, shared variables can be realized as slots in an ancestor object. Here, aSharedVariable is shared by all points, and the global variables point and nil are shared by all objects. This contrasts with Smalltalk, which needs a different linguistic mechanism for class variables and globals.

Consequently, there was no way (until the later development of reflection in Self; see section 4.4) to refer to a method. Many languages exploit references to functions, but Ungar felt that such a facility weakened the object orientation of a language. He felt that since a function always behaves in the same way, unlike an object which can “choose” how to respond to a message, functions-as-first-class entities would be too concrete. In other words, a function called by a function always runs the same code, whereas a method called by a method runs code that is chosen by the receiver. Smith understood Ungar’s reservations about functions but was bothered by the complexity of introducing new fundamental language-level constructs (such as a new kind of slot with special rules for holding methods, or new kind of reference for pointing to a method without firing it).

The reader may wonder how one could ever get a method into a slot in the first place. In the first implementation of Self, the programmer just used a textual syntax to create objects with slots containing data and code. Later, we had to have some way for a program to make new objects and manipulate old ones. The invention of mirrors (section 4.4) added more elegant primitive operations to manipulate slots.

In contrast to many other object-oriented languages including C++ and Java, a number is an object in Self, just as in Smalltalk, and arithmetic is performed by sending messages, which can have arguments in addition to the receiver. For example, $3 + 4$ sends the message `+` to the object `3`, with `4` as argument. This realization of numbers and arithmetic makes it easy for a programmer to add a new numeric data type that can inherit and reuse all the existing numeric code. However, this model of arithmetic can also bring a huge performance penalty, so implementation tricks became especially critical. Self’s juxtaposition of a simple and uniform language (objects for numbers and messages for arithmetic in this case) with a sophisticated implementation let the programmer to live in a more consistent and malleable computational universe.

3.2. Syntax

In settling on a syntax for Self, we automatically borrowed from Smalltalk, as the two languages already had so much in common already. But Self’s use of message sending to replace Smalltalk’s variable access mechanisms would force some differences. Where Smalltalk referenced the class `Point` by having a global variable by that name, Self would reference the prototypical point with a slot named “point” and one would have to send a message, presumably to “self,” to get a reference. So the Self programmer would write

```
sel f poi nt.
```

which was verbose, but seemed acceptable. It raised the uncomfortable issue of what the token “self” meant. Could an object send “self” to itself to get a reference to itself? Smith recalls proposing that every object have a slot called “self” that pointed to itself. But Ungar pointed out that Smith’s proposal only put off the problem one level, as even with the slot named “self,” one would have to send the message “self” to something to get that reference! Smith counterproposed that perhaps there could be an implied infinity of self’s in front of every expression, just as in spoken language, one can say “X” or one can say “I say: ‘X’,” or even “I say ‘I say ‘X’”, and so on. In spoken language we don’t bother with this addition of “I say...” as it goes without saying. One could imagine an infinite number of them in front of any spoken utterance, and that they are just dropped to make spoken language tractable. However, Smith could never formalize this into a working scheme. So, as in Smalltalk, “self” would be a built-in token, providing the self-reference reference ex nihilo.

But that wasn’t the end of it. In a method to double a point,³ the Smalltalk programmer would assign to the two instance variables

```
x ← x * 2.  
y ← y * 2.
```

whereas the Self programmer would perhaps write:

```
sel f x: (sel f x * 2).  
sel f y: (sel f y * 2).4
```

This was getting a bit verbose. One day at PARC, in one of the early syntax discussions, Ungar suggested to Smith that the term “self” be elided. Smith remembers this because he was embarrassed that he had to ask Ungar for the definition of the word “elide.” Ungar explained that it meant the programmer could simply leave out “self.” Under the new proposal, our example became:

```
x: (x * 2).  
y: (y * 2).5
```

At first hesitant, Smith came to like this as dropping the “self” was like dropping the utterance “I say” in natural language. Furthermore, eliding “self” neatly solved the infinite recursion problem of an object’s having to send “self” to self to create a self-reference. In retrospect we feel that was a brilliant solution to a deep problem; at that time, it just seemed weirdly cool.

At this point, readers familiar with C++ will be wondering what the fuss was all about. It is true that C++ unifies the syntax for calling a member function of the receiver with that of calling a global function. Moreover, it unifies the syntax for reading a variable in the receiver with that of reading a global variable, and it unifies the syntax for assigning to a variable in the receiver with that of assigning to a global variable. In summary, C++ has six separate operations that mean six separate things but are boiled down to three syntactic forms: `aFunction(arg1, arg2-----)`, `aVariable`, and `aVariable =`. What we had in Self after eliding “self” was just a single syntax and unified semantics for all six.

3.3. More Semantics

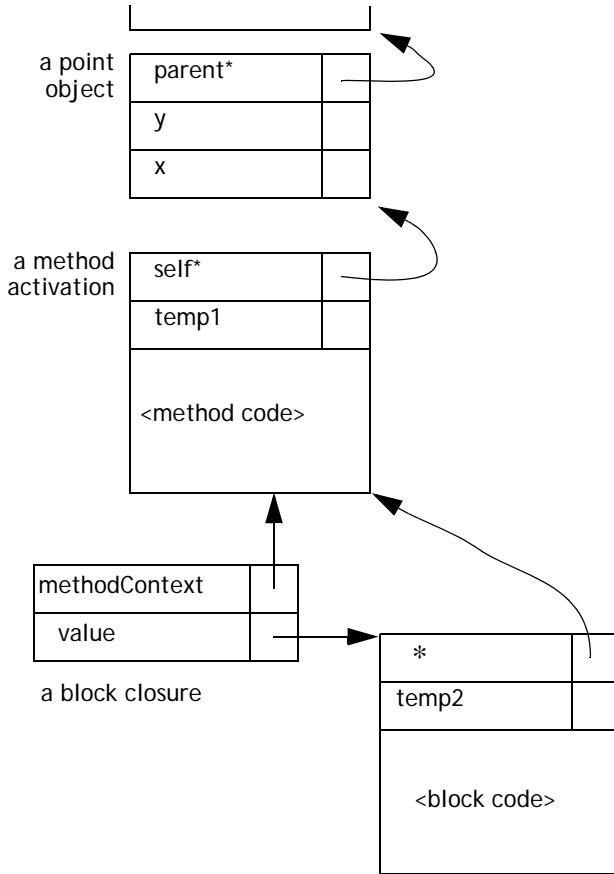
Ungar realized that, having removed variables, he and Smith had stumbled into enshrining message sending as the conceptual foundation of computation. Rather than each expression starting with a variable to serve as some reference, in Self the “programming atom” became a message send. Ungar in particular felt that the syntax could shift people’s thinking about programs so that they would—unconsciously—tend to write better encapsulated and more reusable code. Smith was less interested in syntax, as he felt that whatever reasonable syntax was provided, the underlying semantics would shine through. So, any syntactic realization of the Self computational model would suffice for shifting people’s thinking. Smith therefore felt that since we could choose any reasonable syntax, we should stick with the familiar and thus choose Smalltalk, as it was gaining popularity at the time. Looking back from 2006, Self might have become more popular had we devised a C-style syntax instead.

At this point we still had no good way to deal with temporary variables and arguments, whose scope was limited to a method context. (A method context in Smalltalk or Self is essentially a stack frame, a.k.a. an activation record.) Smith came up with the

3. One has to wonder how the language would have turned out without Cartesian point objects as fodder for our examples.

4. Parentheses added for clarity.

5. Parentheses added for clarity.



idea that rather than retaining Smalltalk's temporaries and method arguments as variables, they too should be slots in an object whose parent was the message receiver. This formulation implied that slot name lookup would start in the local method context and then pass on to the message receiver, and so on up the inheritance hierarchy. Consequently, lookup would not really start at "self," but rather at something like Smalltalk's "thisContext," a pseudo-variable that serves as a reference to the local method activation. Smith explained this to Ungar in Smith's office at Xerox PARC and sensed that though Ungar felt this was a wild idea, he also felt it was somehow right. (Figure 6 illustrates this point.)

Although this tap dance removed the last vestige of variables from the execution story, it left a complexity that bothers us to this day. In Self, everything is a message send that starts looking for matching slots in the current method context, then continues up through the receiver (*self*) and on up from there (see Figure 6). But any method found in the lookup process creates a new method context inheriting from *self*, not from the current context. It's as though the virtual machine has to keep track of two special objects to do its job: the current context to start the lookup, and "*self*," to be the inheritance parent of new activations. Smith wondered how bad it would be to install new activations as children of the current activation, so "*self*" would no longer be such a special object, but Ungar convinced him that the resulting interactions between activations would amount to dynamic scoping and would be likely to create accidental overrides, with confusing and destructive side effects.

Block closures within a method can be represented as objects as well, as also illustrated in Figure 6. When invoked, a block closure is lexically able to refer to temporaries and arguments in its

Figure 6. Lexical scoping of method activations and block closures via inheritance. We were pleased that the lexical scoping rules of methods and block closures could be explained through inheritance. But doing so made us realize there is a fundamental distinction between *self* (which is essentially a parent of the current method activation) and the point at which method lookup starts (which is the activation itself, so that temporaries and arguments in that activation are accessed). In this example, the method code can mention *temp1* as well as *x* and *y*, as message sends start with the current activation and follow up the inheritance chain. But new method sends to *self* will have their *self* parent slot set to the point object.

As detailed in the text, when a block closure is invoked, the closure's activation is cloned, and the implicit parent is set to the enclosing method activation. This link is broken when the enclosing method activation returns.

Thus in the case illustrated here, the code in the block can access *temp2*, *temp1*, *self*, *x*, *y*, *parent*, and any other slots further up the inheritance chain.

enclosing method, but is itself an object that can be passed around without evaluation if desired. In Self or Smalltalk, a block closure can be sent the *value* message to run its code. The *value* method in a block context differs from other methods: when such a method runs its parent slot is set, not to the current receiver, but rather to the enclosing context in which the block originates.

Although the Self model enabled inheritance and slot lookup to explain what many other languages didn't even bother to explain with the language's fundamental semantics, the appearance of special cases (such as the *value* method in a block) bothered us. We had several discussions at the whiteboards at PARC, trying to figure out a unifying scheme, but none was satisfactory.

As we strove for more and more simplicity and purity, we came up against other limits we simply could not wrestle into a pristine framework. We wanted every expression in Self to be composed of message sends. In particular, we wanted every expression to start off by sending one or more messages to the current context and on up through *self*. Literals, though, fail to conform: a literal is an object (usually one of just a few kinds, such as numbers and strings) that is created in place in the code just where it is mentioned. For example, the Self expression

`x sqrt`

sends the message *x* to *self*, then sends *sqrt* to the result. For a few weeks during our design phase we puzzled over how to support the expression

`3 sqrt`

within a pure message-sending framework. Most languages would treat the 3 as a "literal" (something that is not the result of

computation but rather “literally” interpreted directly in place). As an example, in Smalltalk this code fragment would be compiled so as to place the object 3 directly in the expression, followed by the send of `sqrt` to that object. We wondered if Self could treat the textual token 3 not as a literal, but rather as a message send to `sel f`. That way, 3 would be on the same footing as other widely referenced objects, such as `list`, the prototypical list. The idea was that somewhere in the stratosphere of the inheritance hierarchy would be an object with a slot whose name would be “3” and that would contain a reference to the actual object 3. Send “3” to an object, and the result of the lookup mechanism would be a reference to the object that, in the receiver’s context, meant 3. This result would normally be the regular 3, but it might, for example, be a 1 in the context of some object that could only understand mod 2 arithmetic. In the end we gave up on this, as it seemed to hold too much potential for mischief and obfuscation. For instance, $2 + 2$ could evaluate to 5! It seemed to both of us like more expressive freedom than was really needed, and supporting those objects with such a conceptually infinite number of slots seemed a heavy burden to place on the virtual machine. We decided to give up on pushing uniformity this far.

Our design for the unification of assignment with message sending also troubled us a bit. An object containing a slot named “x” that is to be assignable must also contain a slot named “x:” containing a special object known as the assignment primitive. This slot is called an assignment slot, and it uses the *name of the slot* (very odd) to find a correspondingly named data slot in the same object (also odd). This treatment leads to all sorts of special rules; for instance, it is illegal to have an object contain an assignment slot without a corresponding data slot, so consequently the code that removes slots is riddled with extra checks. Also, we were troubled that it took a pair of slots to implement a single container. Other prototype-based languages addressed this issue by making a slot-pair an entity in the language and casting an assignable slot as such a slot pair. Another alternative might have been to make the assignment object have a slot identifying its target, so that in principle any slot could have served as an assignment slot for any other.

Both authors strove for simplicity, but each had his own focus. Smith’s pure vision grounded in the uniformity of the physical world led him to advocate such interesting features as parent slots for methods and message-passing for local variable access. In contrast, Ungar couldn’t wait to actually use the language, and so he was thinking about the interaction between language features and possible implementation techniques. For example, unlike Lieberman’s prototypes [Lieb86, SLU88], a Self object does not add an instance variable on first assignment, but rather must already contain a data- and assignment-slot pair if the assignment is to be local. Otherwise, it delegates the assignment (which is just a one-argument message send) to its parent(s) (if any). Ungar also was thinking about customization (section 4.1) at that point; to make instance variable access and assignment efficient when a sibling might implement them as methods, Ungar realized that one could compile multiple versions of the same inherited method for each “clone-family.” The requirement that an assignable slot be accompanied by a corresponding assignment slot created a clear distinction at object creation time between a constant slot and a mutable slot that was intended from the start to aid the implementer. Ungar knew that an efficient implementation would have to put information shared across all clones of the same prototype in a separate structure, which was eventually called a *map* [CUL89]. (See section 4.1 for details.)

When Ungar came up to PARC as a consultant, he had to sign in by writing his name on an adhesive name tag and wearing it while on the premises, yet no one ever paid any attention to it. So Ungar took to writing more and more absurd names on his tag, such as “nil,” “super,” and even “name tag.” One day, he came into the common area outside Smith’s office at PARC, and upon seeing Smith immediately exclaimed, “I have a name for the language! Self!” He had moments earlier signed his name tag “self” when inspiration had struck. Smith commented that all those selves missing from the syntax could maybe be inherited from the title of the language. The name appealed to us immediately, and from that day forward we had no doubt that the language would be called “Self.”

4. Self Takes Hold at Stanford and Evolves

In June 1986, Ungar (at Stanford) asked Sun for some equipment: an upgrade to 4Mb of main memory for 14 machines (\$28K), a Sun 3/160S-4 workstation with 4MB for (\$15K)—this was a diskless machine—a 400MB disc drive (\$14K), a tape drive (\$3K), and an Ethernet transceiver (\$500). When we started the effort to build a Self system, hardware was primitive and expensive!

Ungar recalls spending his first year at Stanford (1985-1986) casting about for a research topic. His first PhD student, Joseph Pallas, was working on a multiprocessor implementation of Berkeley Smalltalk [Pal90] for the Digital Equipment Corporation Systems Research Laboratory Firefly [TSS88], an early coherent-memory multiprocessor. As far as we know, this system was the first multiprocessor implementation of Smalltalk.

In Ungar’s June 1986 summary of his first year’s research at Stanford, Self was not mentioned at all. But nine months later, he had found his topic: in a March 1987 funding proposal, Ungar wrote: “Self promises to be both simpler and more expressive than conventional object-oriented languages.” He also wrote about “developing programming environments that harness the power of fast and simple computers to help a person create software,” of “transforming computing power into problem-solving power,” of “shortening the debug, edit, and test cycle,” and how “dynamic typing eases the task of writing and changing programs.” He explained the potential advantages of Self: its unification of variable access and message passing, that any Self object could include code and function as a closure, its better program-structuring mechanisms, including prototypes. Finally, he noted that obtaining performance for Self would pose a challenge.

In 1988, Smith went to England for a year, and Ungar’s consulting assignment at PARC changed from designing languages to implementing automatic storage reclamation for what was to become the HPS Smalltalk system. This was the end of Self at PARC.

Ungar had decided that Self’s replacement of variable access by message passing made it so impractical that devising an efficient implementation of Self would make a good research topic. He was also eager to see if the language design would hold up for nontrivial programs. Ungar’s May 1988 report, “SELF: Turning Hardware Power into Programming Power,” proposed a complete, efficient Self virtual machine, a Self programming environment with a graphical interface based upon artificial reality, and a high-bandwidth, low-fatigue total immersion workstation. (We never got around to the last one.) He discussed the benefits of the language and the special implementation challenges it posed. To tackle the implementation issues, we proposed custom compilation and inlining of primitive operations and messages sent to constants. (These were the first optimizations we tried.)

We proposed to investigate dynamic inheritance and to explore mirror-based reflection, the latter as a means to inspect a method as well as objects intended only to provide methods for inheritance by other objects. Ungar christened the latter sort of object a “traits” object. Years later, others would formulate a framework for combining bundles of methods in a class-based framework, reusing the word “traits” with a slightly different meaning [SDNB03].⁶

The report went on to outline proposed work on a graphical programming environment designed to present objects as physical hunks of matter—objects, not windows. Ungar proposed to use well-defined lighting for reality and substance, and to manage screen updates so as to avoid distraction. (This seems to foreshadow our work in cartoon animation.) He also proposed to implement a graphical debugger for Self.

In January 1989, Ungar asked Sun for more equipment: SPARC machines, a server, a workstation for his home, and three diskless machines, each with 24Mb of memory. He also asked for a fast 68030 machine. As the Stanford Self project progressed in 1989 and 1990, it was able to start work on the programming environment and user interface, later known as UI1 or Seity. The goal (far from realized) was eventually to replace most uses of C and C++ (and, of course, Smalltalk) for general-purpose programming. We wanted to harness the ever-increasing raw computational power of contemporary workstations to help programmers become more productive.

4.1. Implementation Innovations

Faced with designing an interpreter or compiler to implement a language, one often takes a mathematical, mechanistic view and focuses on getting correct programs to execute correctly. Inspired by Smith’s Alternate Reality Kit [Smi87], Ungar took a different tack: he concentrated on getting the user to buy into the reality of the language. Even though Self objects had no physical existence and no machine was capable of executing Self methods, the implementation’s job was to present a convincing illusion that these things did exist. That is why, despite all the convoluted optimizations we finally implemented, the programmer could still debug at the source level, seeing all variables while single-stepping through methods, and could always change any method, even an inlined one, with no interference from the implementation.

To structure complexity and provide the freest environment possible, we layered the design so that the Self language proper included only the information needed to execute the program, leaving the declarative information to the environment. In other words, in Smalltalk and Java, classes served as both structural templates (i.e., concrete types) and were visible to the programmer, but in Self the structural templates (embodied by *maps*) were hidden inside the virtual machine and thus invisible to the programmer. Abstract types were regarded as programmer-visible declarative information, and Self left those to the environment. For example, one language-level notion of abstract type, the *clone family*, was used in the work of Ageson et al. [Age96] in their Self type-inference work. There is no clone family object in the Self *language*, but such objects could be created and used by the programming environment. This design kept the language small, simplified the pedagogy, and allowed users to extend the domain of discourse.

The Stanford Self team believed that performance would be critical for acceptance, yet our design philosophy placed a large

burden on the compiler. Deutsch and Schiffman’s PS system had simply translated Smalltalk’s bytecodes to machine code, with only peephole optimization [DS84]. But unlike Smalltalk, the Self bytecodes contained no information about what is a variable access or assignment vs. a message send, and there are no special bytecodes for simple arithmetic, nor special bytecodes for commonly used control structures. Simple translation would not suffice. Obtaining performance without sacrificing the programming experience would be our challenge.

The problem in this area was a magnification of one faced by a Smalltalk implementation: a style of programming in which methods are short, typically one to five lines of code, resulting in frequent message sends. Frequent sends hurt performance because each method invocation in Smalltalk (and Self) is dynamically dispatched. In other words, every few operations a Smalltalk program called a subroutine that depended on the runtime type of the value of the first argument (a.k.a. the receiver). In Self, the situation was even worse, because every variable access or assignment also required a message send. Other, more static languages, such as C++ (and later Java), incorporated static type-checking, and this added information facilitated use of dispatch tables (a.k.a. vtables) to optimize virtual calls. This technique was not suited for Self or Smalltalk because without static types every dispatch table needs an entry for every method name. This requirement would result in prohibitive time and space requirements to update and maintain dispatch tables. Thus, to make Self work well, we would not only have to implement prototypes effectively, but would also have to find new techniques to eliminate the overhead of virtual function calls by inline expansion of methods whose bodies could not be known before the program runs.

4.1.1. The First Self Virtual Machine, a.k.a. Self-89

Back when the language had been designed, Ungar had deviated from Lieberman’s prototype model for implementation considerations. In Lieberman’s system, an object initially inherited all of its attributes and gained private attributes whenever an assignment occurred. From the beginning, Ungar tried to keep an object’s layout constant to reduce run-time overhead. He therefore incorporated the distinction between variable and constant slots into Self. Assignment could only change a variable slot, not create a new slot. Furthermore, to represent an object, space would be required for only its variable slots and one pointer to shared information about its constant slots and its layout. This shared information was called a map (Figure 7). During 1986-87, graduate students Elgin Lee and Craig Chambers joined the project. Lee wrote the first memory system and implemented maps to achieve space usage that was competitive with Smalltalk [Lee88]. Later, Chambers reimplemented the memory system [CUL89]. We achieved our goal: the per-object space overhead of Self was only two words.

In 1988, Chambers wrote the first Self compiler [CU89, CUL89]. This compiler represented Self programs using expression trees and introduced three techniques: customization, type prediction, and message splitting. Each of these ideas was inspired by our desire to run no more slowly than Smalltalk. Wherever we thought that Self’s object model would hinder its performance, we tried to devise a technique to recoup the loss, at least in the common cases. To maintain the interactive feel of an interpreter, we also introduced dependency lists (described below).

Customization. A Smalltalk object belongs to a specific class, and its instance variables occur at fixed offsets specified by the class. Even an inherited instance variable has the same offset as

6. According to an email exchange with Black and Schärli, Self’s traits played into their thinking but were not the primary inspiration.

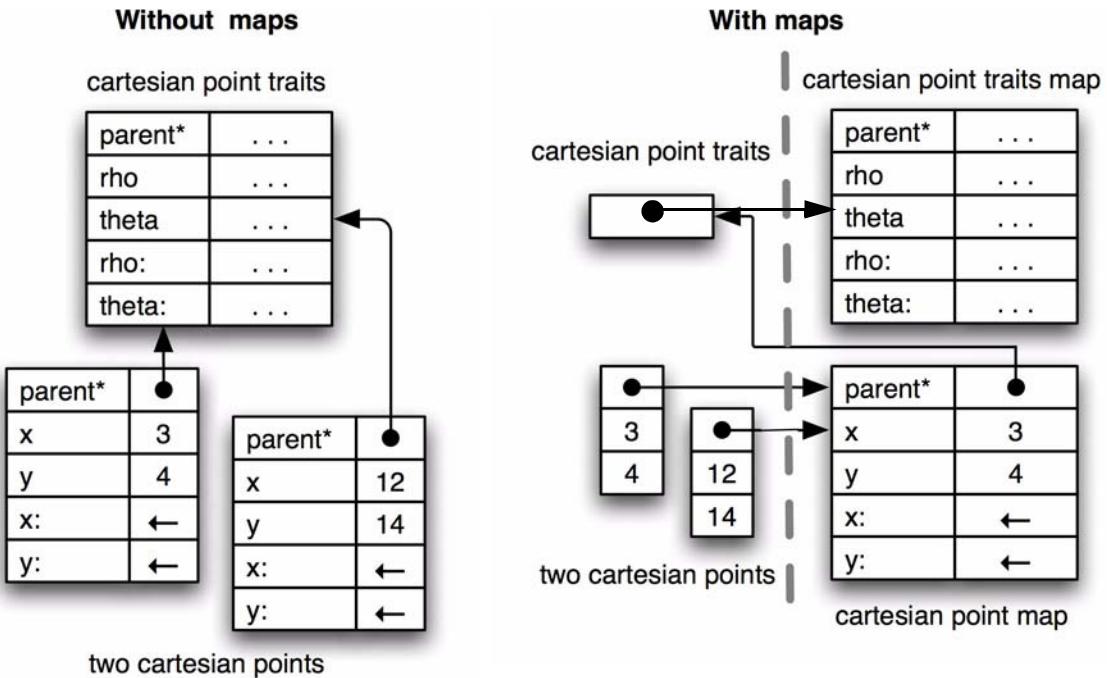


Figure 7. An example of the representations for two Cartesian points and their parent, also known as their “traits” object. Without maps, each slot would require at least two words: one for its name and another for its contents. This means that each point would occupy at least 10 words. With maps, each point object needs to store only the contents of its assignable slots, plus one more word to point to the map. All constant slots and all format information are factored out into the map. Maps reduce the 10 words per point to 3 words. (A Self object also has an additional word per object containing a hash code and other miscellany.) Since the Cartesian point traits object has no assignable slots, all of its data are kept in its map.

it would in an instance of the class from which it is inherited. As a result, the Smalltalk bytecodes for instance variable access and assignment can refer to instance variables by their offsets, and these bytecodes can be executed quite efficiently. In Self, there is no language-level inheritance of instance variables, and so the same inherited method might contain an access to an instance variable occurring at different offsets in objects cloned from different prototypes. (All objects cloned from the same prototype have the same offsets, and are said to comprise a clone family.) In some invocations of the inherited method, the bytecode might even result in a method invocation. To compile accesses as efficient, fixed-offset load operations, Ungar had realized—back at PARC—that the virtual machine could compile multiple copies of the same inherited method, one per clone family. This trick would not compromise the semantics of the language because it could be done completely transparently. This technique, known as customization, was implemented in Chambers’ first Self compiler (see Figure 8).

Type Prediction. In Smalltalk and Self, even the simplest arithmetic operations and control structures were written as messages. In the case of control structures, blocks are used to denote code whose execution is to be deferred. Thus, even frequently occurring operations that need not take much time must be expressed in terms of general and relatively time-consuming operations. For example, the code `a = b ifTrue: [. . .]` sends a message called “=” to `a`, then creates a block, and finally sends “`ifTrue:`” to the result of “=” with a block argument. The Smalltalk system uses special bytecodes for arithmetic and simple control structures to reduce this overhead. For Self, we kept the bytecode set uniform, but built heuristics into the compiler to expect that, for instance, the receiver of “=” would probably be a (small) integer and that the receiver for “`ifTrue:`” would

likely be a Boolean. This information allowed the compiler to generate code to test for the common case and optimize it as described below, without loss of generality. For example, in Self but not Smalltalk, the programmer can redefine what is meant by integer addition. This idea was called “type prediction.”

Message Splitting. As mentioned above, Smalltalk implemented if-then constructs such as `i fTrue:` with specialized bytecodes, including branch bytecodes. Since in Smalltalk (and Self) the “true” and “false” objects belong to different classes (clone-families in Self), the branch bytecodes conceptually test the class of the receiver to decide whether to branch or not. To achieve similar performance in compiled Self code without special bytecodes, we had allowed the compiler to predict that the receiver of such a message was likely to have the same map as either the “true” or the “false” object, but could be anything. The Self compiler was built around inlining as its basic optimization, so to optimize `i fTrue:` for the common case without losing the ability for the user to change the definition of `i fTrue:`, the compiler had to insert a type-case (a sequence of tests that try to match the type (represented in Self by the map) of the receiver against a number of alternatives) and then inline different versions of the called method in each arm of the type-case construct. In the “true” arm, it could inline the evaluation of the “then” block, in the “false” arm, it could inline “nil,” and in the uncommon case, it could not inline at all but just compile a message-send. In other words, one message-send of “`ifTrue:`” was split into three sends of “`ifTrue:`” to three different types of receiver (true, false, and unknown). We dubbed this technique “message splitting.”

Dependency Lists. For this compiler Chambers also created Self’s dependency system, a network of linked lists that allowed the virtual machine to quickly invalidate inline caches and com-

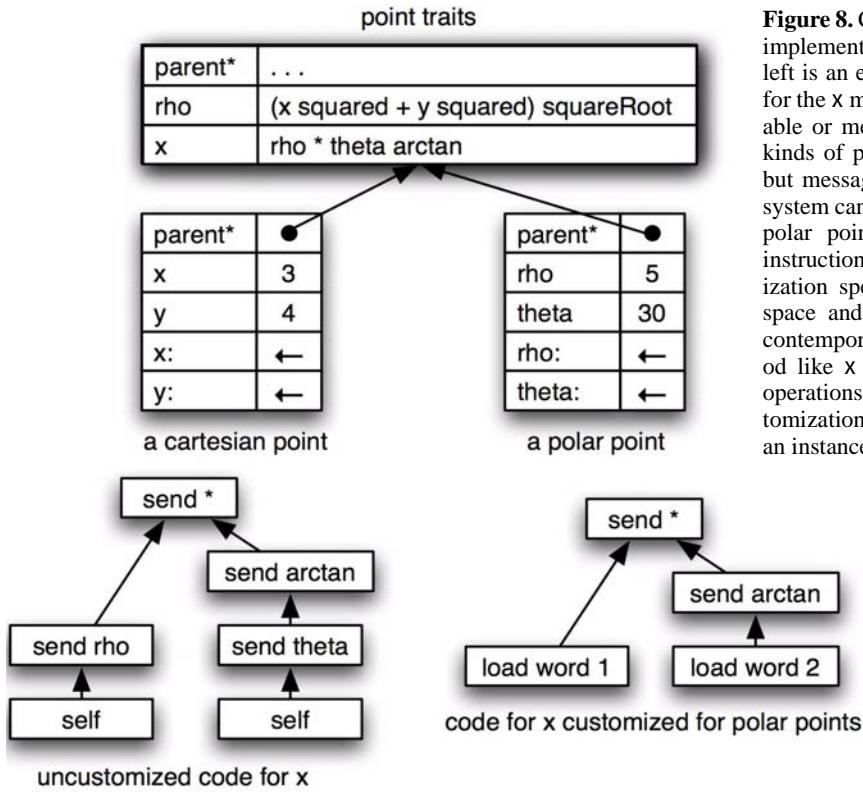


Figure 8. Customization: At left are three objects implementing Cartesian and polar points. Below left is an expression tree for uncustomized code for the x method. Since rho may be either a variable or method, to use the same code for both kinds of points, nothing more can be compiled but message sends for rho and theta. But if the system can compile a specialized version of x for polar points, it can replace these with load instructions, as in the code below right. Customization speeds sends to self at the expense of space and complexity. In Smalltalk and other contemporary object-oriented languages, a method like x could include instance variable load operations at its source level. Ungar devised customization to regain the speed lost by expressing an instance variable access as a send to self.

piled methods when the programmer made changes to objects [Cham92]. In addition to these techniques, the compiler also supported source-level debugging so that the system could be as easy to understand as an interpreter.

When this compiler was completed in 1988, the Self virtual machine comprised 33,000 lines of C++ and 1,000 lines of assembly code. It ran on both a Motorola 68020-based Sun-3 workstation and a SPARC-based Sun-4 workstation. The latter was a RISC microprocessor with a 60ns cycle time, and an average of 1.6 cycles per instruction. We had written about 9,000 lines of Self code, including data structures, a simple parser, and the beginnings of a graphical user interface. The largest benchmark we used at that time was the Richards operating system simulation [Deu88], and the compiler produced Self code that ran about three times faster than Smalltalk, but about four times slower than optimized C++ [CU89]. There was an issue with compilation speed: on a Sun 4-260 workstation, the compiler took seven seconds to compile the 900-line Stanford integer benchmarks, and three seconds to compile the 400-line Richards benchmark. This was deemed too slow for interactive use; we wanted compile times to be imperceptible.

Ungar recalls that Chambers articulated an important lesson about types: the information a human needs to understand a program, or to reason about its correctness, is not necessarily the same information a compiler needs to make a program run efficiently. Thereafter, we spoke of *abstract types* as those that help the programmer to understand a program and of *concrete types* as those that help an implementation work well. In many languages, the same type declaration (e.g., 32-bit integer) specifies both semantics and implementation. As we implemented Self, we came to believe that this common conflation inevitably compromised a language's effectiveness.⁷ Accordingly, we hoped to

show that type declarations for the sake of performance were a bad idea, and we made the point that Self's performance—without explicit declarations—had already pulled even with Johnson's Typed Smalltalk system [John88, CU89].

4.1.2. The Second-Generation Self Virtual Machine, a.k.a. Self-90

We weren't satisfied with the performance of our first Self compiler and in 1989 proceeded to improve the Self system. In early 1989, Chambers rewrote the memory system and then implemented a far more ambitious compiler [CU90]. This compiler was based on a control flow graph and included many optimization techniques that had been invented for static languages, such as more extensive inlining, interprocedural flow-sensitive type analysis, common subexpression elimination, code motion, global register allocation, instruction scheduling, and a new technique called extended splitting.

Extended Splitting. Recall that the first Self compiler had been based on expression trees. As a consequence, the only message sends that it could split were those whose receivers were the results of the immediately preceding sends. With the addition of flow-sensitive type analysis, the new compiler could split a message based on the type of a value previously stored in a local variable. We observed that it was common for the same local to be the receiver for several message sends, although the sends might not be contiguous, so Chambers extended the new compiler to split paths rather than individual sends. This technique was called “extended splitting” and the ultimate goal was to split off entire loops, so that, for example, an iterative calculation

7. The creators of the Emerald system had the same insight [BHJ86] and had probably discussed it with Ungar during a visit to the University of Washington.

involving (small) integers could be completely inlined with only overflow tests required. Many of the benchmarks we were using then consisted of iterative integer calculations because we were trying to dethrone performance champion C, and those sorts of programs catered to C's strengths.

The new compiler yielded decidedly mixed results. The performance of the generated code was reasonable: the Richards benchmark shrank to three-fourths of its previous size and slowed by just a bit, small benchmarks sped up by 25% - 30%, and tiny benchmarks doubled in speed. The problem was that compiler ran an order of magnitude slower. For example, it took a majestic 35 seconds to compile the Richards benchmark, a wait completely unsuitable for an interactive system. Perhaps we had fallen prey to the second-system syndrome [Broo]. In day-to-day use, we stuck with the original compiler.

This second-generation system did introduce other improvements, including faster primitive failure, faster cloning, faster indirect message passing (for messages whose selectors are not static), blocks that would not crash the system when entered after the enclosing scope had returned, and a dynamic graphical view of the virtual machine (the “spy”), written by Urs Hözle, who had joined the Self project in the 1987-1988 academic year.

4.1.3. The Third-Generation Self Virtual Machine, a.k.a. Self-91

By mid-1990, Hözle had made many small but significant improvements to the Self virtual machine: he had improved the performance of its garbage collector by implementing a card-marking store barrier; he had redone the system for managing compiled machine code by breaking it up into separate areas for code, dependencies, and debugging information; he had added an LRU (least-recently used) machine-code cache replacement discipline; he had started on a profiler; and he had improved the method lookup cache. The Self virtual machine comprised approximately 50K lines of C++.

Hözle had devised a new technique that would turn out to be crucial: polymorphic inline caches (PICs) [HCU91]. Self was already using inline caching [DS84], a technique that optimized virtual calls by backpatching the call instruction. Deutsch and Schiffman had noticed that most virtual calls dispatched to the same function as before, so rather than spending time on a lookup each time, if the call went to the same method as before, the method could just verify the receiver's map in the prologue and continue. This technique worked, but we discovered that some fairly frequent calls didn't follow this pattern. Instead, they would dispatch to a small number of alternatives. To optimize this case, when a method prologue detected an incorrect receiver type, Hözle's new system to create a new code stub containing a type-case and redirect the call instruction to this stub. This type-case stub, called a polymorphic inline cache (PIC), would be extended with new cases as required. This optimization sped up the Richards benchmark, which relied heavily on one call that followed this pattern, by 37%. We realized that, after a program had run for a while, the PICs could be viewed as a call-site-specific type database. If a call site was bound to the lookup routine, it had never been executed; if it was bound to a method, it had been executed with only one type; and if it was bound to a PIC, that PIC contained the types that had been used at that site. Hözle modified Chambers' compiler to exploit the information recorded in the PICs after a prior run and sped up Richards by an additional 11%.

In 1990, Chambers worked to improve the compilation speed without sacrificing run-time performance [CU91]. We had learned that much published compiler literature neglected the

compilation speed issue that was so critical to the interactive feel we wanted for Self. Striving for the best of both worlds, Chambers devised a more efficient implementation of splitting and enhanced the compiler to defer the compilation of uncommon cases. The latter idea was suggested to us by then-student John Malone at an OOPSLA conference (Malone would later join the Self project). Deferred compilation avoided spending time on the cases that were expected to be rare, such as integer overflow and out-of-bounds array accesses. The compiler still generated a test for the condition, but instead of compiling the rarely executed code, would compile a trap to make the system go back and transparently recompile a version with the code included for the uncommon case. The new version would be carefully crafted to use the same stack frame as the old, and execution would resume in the new version. The whole process was (naturally, given our proclivities) transparent to the user.

In addition to hastening compilation, this optimization sped up execution because the generated methods were smaller and could use registers more effectively. However, in subsequent years, it turned out to be a source of complexity and bugs. As of this writing (2006), Ungar, who has only his spare time available to maintain Self, has disabled deferred compilation. Back in 1990, though, we were excited: the system compiled the Richards benchmark 7 times faster than previously, the compiled code was about three-fourths the size, and it ran 1.5 times faster. This brought Richards performance to one third that of optimized C++. We released this system as Self 1.1 in January 1991.

With all the improvements, compilation speed on our Sun 4-260 was still too slow; compiling Richards took 5.5 seconds. In addition, this third compiler suffered from brittle performance; because it used heuristics to throttle its inlining, it was sensitive to the program's exact form, and small changes to a program could result in large changes to its performance, as method sizes crossed inlining thresholds. However, after three compilers, it was time for Chambers to stop programming and start his doctoral dissertation. He did so and graduated in 1992.

It was then Hözle's turn to take on the challenge of combining interactivity with performance. Building on Chambers' compilers and his own work with polymorphic inline caches, he started to experiment with “progressive compilation” and would eventually achieve the best of both worlds (section 5.1).

4.2. Language Elaborations

In 1988 and 1989, the students and Ungar writing Self code at Stanford ran into situations that seemed to need better support for multiple inheritance and encapsulation than were covered by the language outlines as sketched out at Xerox PARC. Self's simple object model was a good base for exploring these topics since there were few interactions with other language features. Smith was following other research interests at this point, and so Chambers, Ungar, Chang, and Hözle set about enhancing the language with some clever ideas: prioritized multiple inheritance, the sender-path tiebreaker rule, and parents-as-shared-parts privacy [CUC91].

Prioritized Multiple Inheritance. Back in the late 1980s, multiple inheritance was a popular research area, especially rules for dealing with collisions arising from inheriting two attributes with the same name [Card88]. Class-based languages suffered from the need to deal with structural collisions arising from inheriting different instance variables with the same name, as well as behavioral collisions arising from inheriting different methods with the same name, and we thought that this area would be easier in classless Self. There were two popular search

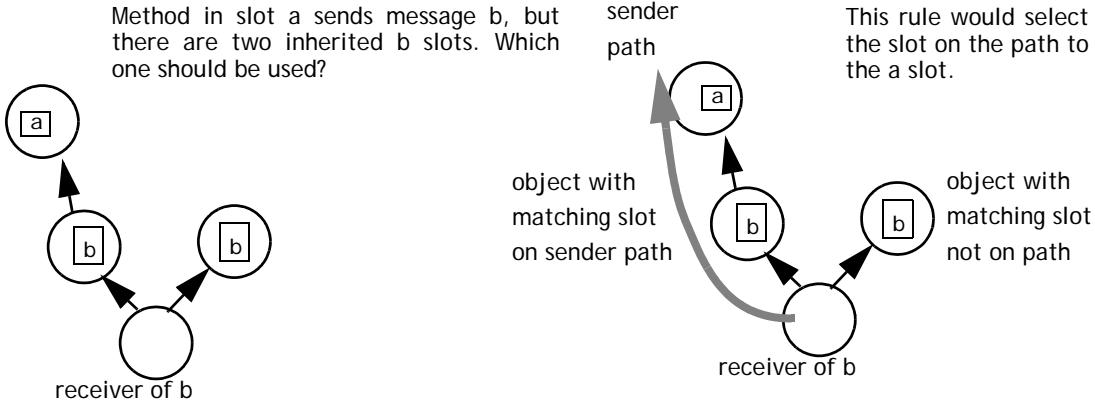


Figure 9. Sender path tiebreaker rule. For a while, multiple inheritance conflicts were resolved according to the inheritance path of the sending method. In this situation there is a “tie” with two inherited ‘b’ slots. The ‘b’ slot on the left is selected because it is on the path to the slot whose code sent the ‘b’ message.

strategies for multiple inheritance: unordered for safety, and ordered for expressiveness. In the unordered case, all parents were equal and any clashes were errors. In the ordered case, parents were searched in order, and the first match won. Seeking the best of both worlds, the Stanford students and Ungar devised a priority scheme: each parent was assigned an integer priority by the programmer, and the lookup algorithm searched parents in numeric order. Equal-numbered parents were searched simultaneously, and multiple matches in equal-numbered parents generated an “ambiguous message” run-time error.

Sender-path Tiebreaker. Having devised and implemented a powerful multiple inheritance scheme, we set about using multiple inheritance wherever we could. As a result, we struggled with many “ambiguous message” errors in our code. Since most of these errors seemed unjustified, we came up with a new rule that we thought would automatically resolve many of the conflicts. This rule stemmed from our belief that parent objects in Self were best considered to be shared parts of their children. When combined with the typical case of a method residing in an ancestor of its receiver, we believed that a matching slot found on the same inheritance path as the object holding the calling method ought to have precedence. This was called the “sender-path tiebreaker rule” (see Figure 9).

Shared-part Privacy. Smalltalk provides encapsulation for variables but not methods; in Smalltalk, instance variables are private to the enclosing object, but all methods are public. Since we believed that a Self variable should be thought of as just a partic-

ular implementation of two methods, the original design for Self omitted encapsulation for all variables (as well as methods). Influenced by Smalltalk and, to a lesser extent, by C++, the Self group (then at Stanford) sought to fix this by adding privacy to the language. At this time we had yet to build a graphical user interface, and so we started with a discussion of syntax in Ungar’s office that lasted for hours. Eventually, Chambers facetiously proposed an underscore prefix (“_”) for private slots and a circumflex prefix (“^”) for public slots. When Ungar agreed, Chambers tried to unpropose them but failed, and so those prefixes became Self’s privacy syntax. After agreeing on syntax, we then had to devise a semantics for privacy. Consider a slot a containing a method that sends the message b. If b is private, how should we decide whether to allow the attempted access to b found in slot a’s code? Reasoning that in Self, parents are shared parts of their children, we decided that slot b should be accessible to a given message from code in a if both the object holding the a slot and the object holding the b slot were either the *same as* or *ancestors of* the receiver of the message. This concept was called “shared-part privacy” (see Figure 10).

Chambers deftly made these changes to the virtual machine. He did it so easily that back then, Ungar felt that there was no language feature too intricate for Chambers to put into the system in a day or so. Of course, having invented a powerful new privacy scheme, we set about writing programs that put it to work whenever possible.

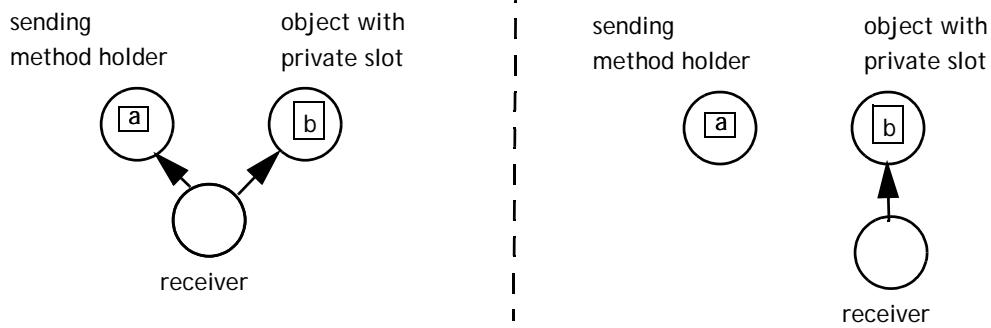


Figure 10. Privacy based on parents-as-shared-parts. Inherited method a sends message b to self, labeled as the receiver. In each case b is a private slot. Since both the sending method holder and the private slot holder are parents of the receiver on the left, that access would be allowed. On the right, the access would be denied.

In a July 1989 report, “SELF: Turning Hardware Power into Programming Power,” the multiple inheritance with send-path tiebreaking idea appeared. Ungar was enthusiastic about this idea at the time: “SELF’s multiple inheritance innovations have improved the level of factoring and reuse of code in our SELF programs.” The first version of the privacy (a.k.a. encapsulation) idea also appeared as a proposal. By the end of March 1990, we had added our new privacy semantics, had changed “super” to “resend” to accommodate “directed resends,” and had changed the associativity rules for binary messages to require parentheses.

Something unexpected occurred after we started using our multiple inheritance and privacy schemes. Over the following year, we spent many months chasing down compiler bugs, only to discover that Chambers’ compiler was correct and it was our understanding of the effects of the rules that was flawed. For example, a “resend” could invoke a method far away from the call site, running up a completely different branch of the inheritance graph from what the programmer had anticipated. In addition, the interactions with dynamic inheritance turned out to be mind-boggling. Eventually, Ungar realized that he had goofed. Prioritized multiple inheritance, the sender-path tiebreaker rule, and shared-part privacy were removed from Self by June 1992. We found that we could once again understand our programs. Self’s syntax still permitted programmer to specify whether a slot was public, private, or unspecified, but there was no effect on the program’s execution. This structured comment on a slot’s visibility proved to be useful documentation.

In the process of revisiting Self’s semantics for multiple inheritance, Chambers suggested that we adopt an “unordered up to join” conflict resolution rule (see Figure 11). Although it might have worked well, we never tried this idea; once bitten twice shy.

To this day, many object-oriented language designers shy away from multiple inheritance as a tar pit, and others are still trying to slay this dragon by finding the “right” concepts. Our final design for Self implemented simple, unordered multiple inheritance and has proven quite workable. Although many language designers (including Ungar) have used examples to motivate the addition of facilities, at least for prioritized multiple inheritance, the sender path tiebreaker rule, and shared-part privacy, it would have been better to let the example be more difficult to express and keep the language simpler. Ironically, in his dissertation,

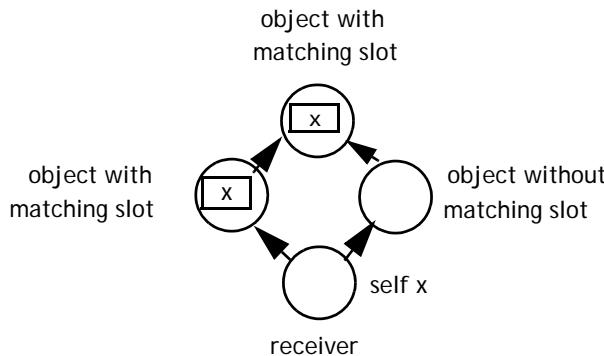


Figure 11. Unordered up to join: Under Chambers’ proposed scheme, there would be no conflict in this case, since the first match precedes a join looking up the parent links. Under our old sender-path tiebreaker, there still could be a conflict if the sending method were held by any but the leftmost object. There is also a conflict under the current rules for Self.

Ungar had written about this danger for CPU designers, christening it “The Architect’s Trap” (section 2.4). On the one hand, some lessons seem to require repetition. On the other hand, maybe we just gave up too soon.

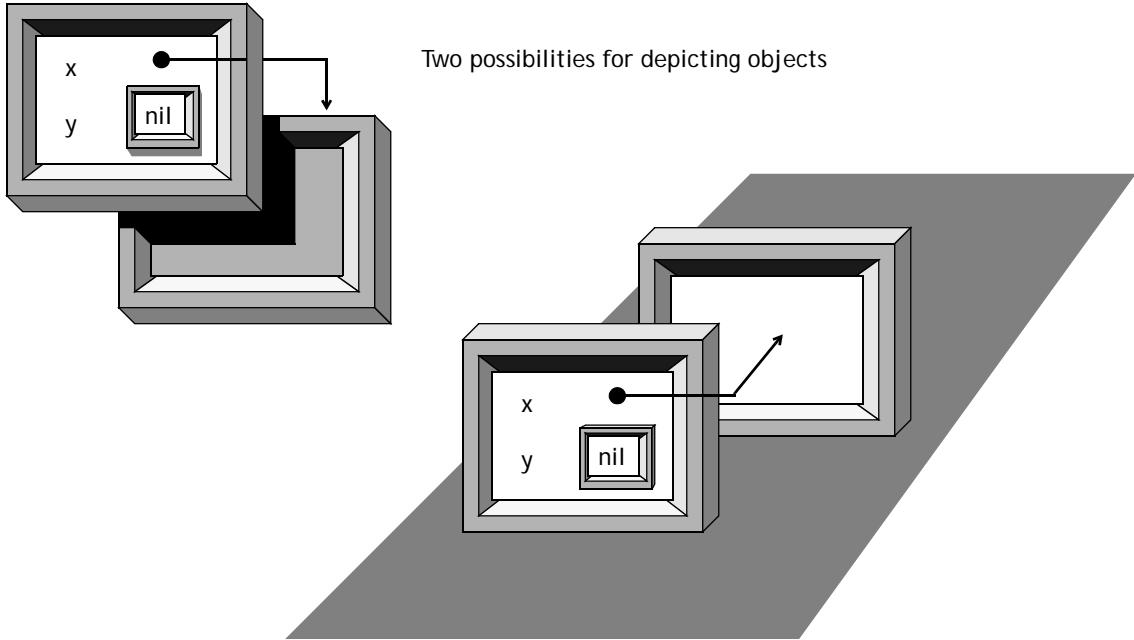
4.3. UI1: Manifesting Objects on the Screen

In the spring of 1988, Bay-Wei Chang, then a graduate student at Stanford, took Ungar’s programming languages class. He became interested in Self and was impressed when, during the final exam for the class, a video tape on the Alternate Reality Kit was shown. This was, in Chang’s own words, “a cruel trick to play, as after the video I sat with my mouth agape for precious minutes.” In the fall of 1988 Chang undertook an independent project working on version 1 of the Self UI, and officially joined the Self project in early 1989. Inspired by the Alternate Reality Kit, Ungar encouraged Chang to craft a user experience that would be more like the consistent illusion of a Disneyland ride than the formal system of a programming language. We wanted to construct the illusion that objects were real (see Figure 12). In May 1989, with the incorporation of Interviews/X and Pixrect primitives into Self, Chang was able to write a mock-up of a direct-manipulation Self user interface. By the end of 1989, this environment was further improved with fast arrowheads, better object labeling, and optimizations that included our own low-level routines to copy data and draw lines. As a result performance improved from 10 to 30 frames/sec. on a monochrome SPARCstation-1.

The original version of UI1 (written by Chang at Stanford in 1988-1989) had run on machines with monochrome frame buffers. By 1990, we had eight-bit frame buffers, although (as Ungar recalls) we had grayscale monitors and there was no hardware acceleration. Ungar realized that, by reducing our palette of colors (actually, grays), we could use colormap tricks to get smooth, double-buffered animation on the screen. We achieved 30 frames/sec. on a color SPARCstation with a graphics accelerator. By the end of one year (May 1990) this version of UI1 was working (see Figure 13).

As of 2006, colormaps have disappeared from most computers, so the reader may not know this term. A “colormap” is simply an array of colors. An image composed of pixels that use a colormap doesn’t store the color information directly in the pixel, but rather stores the colormap array index for that color in the pixel. The key advantage of the colormap for animation effects arises from the simple reality that a window on the screen typically has about a million pixels, whereas a colormap has only 256 entries. Thus, a computer can run through this very short color map, changing the colors stored in various indices, to obtain a nearly instantaneous visual effect on millions of pixels. Consider a colormap with the color white stored at both index 0 and index 128. A screen image with pixel data that is all 0 except for a region with 128 appears to the user as entirely white. But when a program stores the color black at colormap index 128, a black region suddenly appears on the white background.

Suppose that a 256-color colormap is split into four identical parts, so that every entry from 0 through 63 is replicated three more times through the colormap indices. This limits the range of available colors to 64, but it frees up two “bit planes” for drawing: setting bit 7 in a pixel’s value (effectively adding 128 to the data for that pixel) has no visual effect. Nor does setting bit 6. Suppose the colormap is suddenly modified so that all indices with bit 7 set to 1 are black. Black regions will instantaneously appear on the screen wherever pixels have values with



Two models for a syntax tree data structure

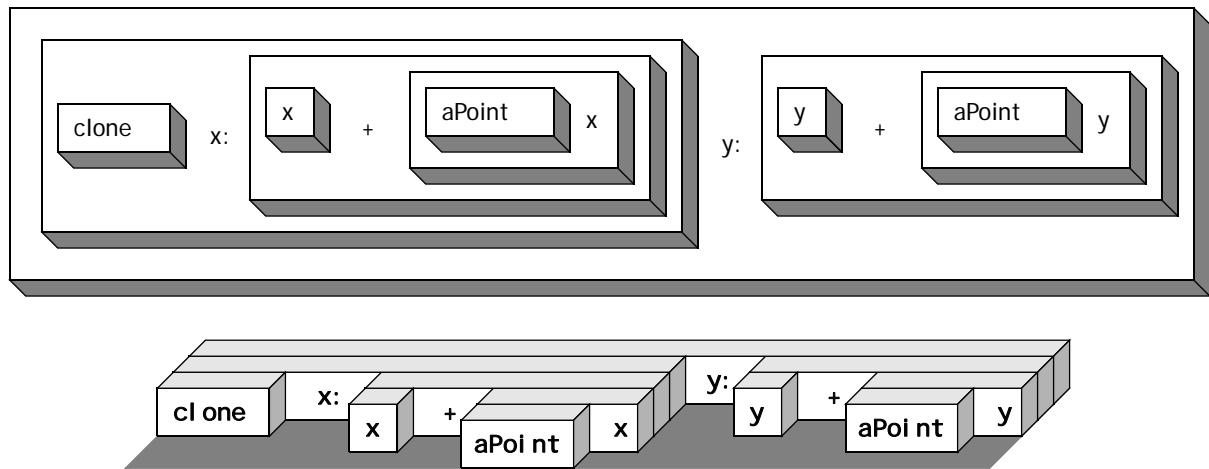


Figure 12. Visions of a Self user interface taken from a May 1988 grant proposal. Above, two possibilities for objects; below, two possibilities for a syntax tree. From the proposal: “We are interested in pursuing a style of interaction that can exploit what the user already knows about physical objects in the real world. For this reason, we call this paradigm *artificial reality*. For example, instead of windows that overlay without any depth or substance, we will represent objects as material objects, with depth, lighting, mass, and perhaps even gravity.”

that bit set. Suppose then that the color map is restored except that now bit 6 is set to black. The first set of black regions disappears but a new set of black regions appear. Clearing bit 7 in the image data, drawing with bit 7, then changing the colormap appropriately makes yet another change appear on the screen. Alternately clearing and drawing with first bit 7, then bit 6, animated images can be made to appear over the background. The two bit planes are being used to achieve an animation drawn with one color. Because one plane remains visible while another, invisible plane is used for drawing, this scheme is an instance of what is termed a “double buffering” animation technique.

In UI1, Chang and Ungar carried this technique further by dividing up the frame buffer into two sets of one-bit planes and two sets of three-bit planes. The one-bit planes double-buffered the arrows, and the three-bit planes double-buffered the boxes. (The boxes needed three bits so they could have highlight and shadow colors.) The arrows were separated from the boxes to make it easier to depict the arrows as being in front of the boxes. At any given time, the colormap would be set so that one arrow and one box plane was visible. UI1 would then compute the next frame of arrows and boxes into the invisible planes, then switch the colormaps. Later at Sun, when we added dissolves⁸, we would put each key frame into a separate plane and update the color-

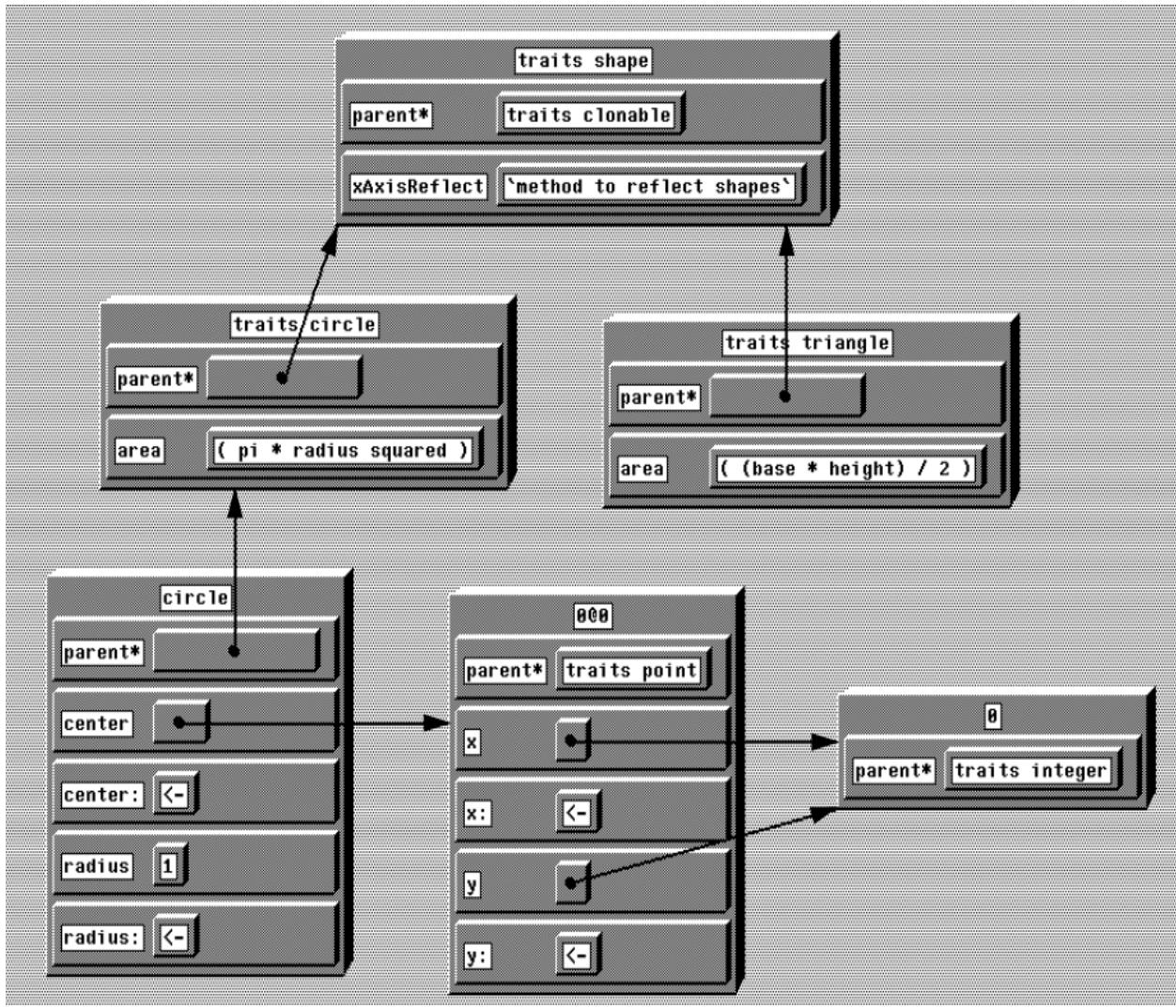


Figure 13. The original Self programming environment, the first version of UI1, was designed to be object-centered. Each box represented a Self object, and a pseudo-3D style attempted to convey a sense of physical reality. (Picture copied from [CU90a].)

map for each frame. This trickery enabled UI1 to display 20 to 30 frames/sec. smooth-looking animation on the hardware of 1991.

At this time, we were writing Self code with a text editor, and feeding the files to a read-eval-print loop. We even built a text-based source-level debugger with commands resembling those of the Gnu debugger, *gdb*. But we knew that eventually we wanted to live in a world of live objects—after all, we were inspired by Smalltalk! However, it was not until UI2 (section 5.3) and the Self transporter (section 5.5) that we could make the change, and even then at least one of the team members, Ole Agesen, still sticks to text editing. At this writing, we asked Agesen to recall why he kept using the older approach: he responded that he was in a rush to complete his thesis, partly in fear that the project would be canceled, so he didn't want to take the time to learn how to transition, nor take the risk of relying on as yet unproven technologies for his thesis work.

8. A “dissolve” is a transition in which one frame smoothly changes into the next. Each pixel slowly changes from its value in the first frame to its value in the new frame.

For UI1, we pushed hard on being object-centered; there would be nothing on the screen (except for pop-up menus) that was not an object. No browsers, no inspectors, just objects. It was the Self *language* that made this a reasonable approach. For example, to understand a Smalltalk program, one must understand the behavior as manifested by the inheritance hierarchy, as well as the state of all the variables in the current scope. The Smalltalk browser could show the inheritance story, but the variable values were held in several objects scattered at conceptually remote places in the system, and viewing them required other tools, such as the “inspector,” unrelated to the Smalltalk inheritance hierarchy. But Self’s use of message passing for variable access meant that the inheritance hierarchy of actual objects was all the programmer needed to see both behavior and state. And, as previously mentioned, to use the Smalltalk-80 browser, one had to learn the role of categories, method protocols, and the instance/class switch as well. But for a Self environment, Chang and Ungar needed only to build a good representation of a Self object, and that would serve most of the programmer’s needs.

Unlike Smalltalk, in which one could have multiple inspectors on the same object, Self’s UI1 allowed only one representation of the object on the screen. We were trying to preserve the illu-

sion that the picture on the screen *was* the object. An object was rendered using a pseudo-3D representation and had a context-dependent pop-up menu. Clicking on a slot would sprout an arrow to its contents. If the referenced object was not already on the screen it would be summoned. If it was already there, the arrow would just point to it. UI1's object-centrism was successful in helping us ignore the artifice behind the objects. About ten years later, we finally decided to compromise this principle because it was so handy to have a collection of slots from disparate objects. For example, one might want to look at all implementers of "display." We called such things *slices*, and included them in our later environment, UI2.

4.4. Reflecting with Mirrors

By unifying state and behavior and eliminating classes, the design of Self encapsulated the structure of an object. No other object could tell what slots some object possessed, or whether a slot stored or computed a result. This behavior is well-suited for *running* programs, but not for *writing* programs, which requires working with how an object is implemented. To build a programming environment, we needed some programmatic way to "look inside" of an object so that its slots could be displayed.

Ungar's first thought was to follow Smalltalk's practice and add special messages to an object to reveal this information. This would also have fit with Smith's idea of emulating physical reality, in which every object is fully self-contained. This architecture proved to be unworkable, since one could not even utter the name of a method object without running it. Ungar reasoned that Self needed a kind of ten-foot pole that would look at a method without setting it off. He used the word "mirror" for the ten-foot pole, both to connote smoke-and-mirrors magic and also to pun on the optical meaning of "reflection."

The mirror behaves like a dictionary whose keys are the names of the object's slots and whose contents are objects representing the slots. To display an object, the environment first asks the virtual machine for a mirror on the object. Following an object's slot through a mirror yields another mirror that reflects the object contained in the slot. In this fashion, once a mirror has been obtained, all of the information encapsulated in the mirror's *reflectee* can be obtained from the mirror. A method is always examined via its mirror, and is thus prevented from firing. By May 1990, we had read-only reflection (a.k.a. introspection) via mirrors.

Once we started thinking about mirrors, other advantages of this architecture became apparent. For example, since only one operation in the system creates a mirror, and since, to the virtual machine, a mirror looks slightly different than an ordinary object, introspection can be disabled by shutting off the mirror creation operation and ensuring there are no existing mirrors. Much later (ca. 2004), we exploited the mirror architecture to implement remote reflection for the Klein project, by implementing an object that behaved like a mirror but described a remote object [USA05].

Also, mirrors were a natural place to support the kinds of changes to objects that a programmer would effect with a programming environment. To minimize the extra complexity in the virtual machine, Ungar borrowed a page from functional programming. With the sole exception of the side-effecting `defi ne` operation, all of the primitive-level reflective mutation operations created altered copies instead of modifying existing objects. For example, when the user changed a method on the screen, the programming environment would have to alter the contents of a constant slot, and this was a reflective operation. However there was no reflective operation that altered a con-

stant slot in place; instead there was a functional operation that produced a new object with an altered slot. After obtaining this new object, the environment would invoke `defi ne`, which would redirect all references from the original object to (a copy of) the new one. This design ensured that only the `defi ne` operation⁹ needed to invalidate any compiled code, since none of the others altered existing objects. The "copy of" was part of the define operation's semantics to optimize this operation when the old and new objects were the same physical size in memory. (In that case, the system could just overwrite the old object with the contents of the new.)

At the time, mirrors seemed merely a good design but not significant enough to publish. This system was working by the end of June 1991, and was used by UI1 to allow the user to change the objects on the screen. The VM was even able to update code that had been inlined, thanks to trapping returns and lazily recompilation. This was a milestone: the graphical programming environment was finally usable for real programming. Years later, Gilad Bracha, who was working at Sun and knew of the Self project work, thought it would be important to generalize and explain this design, and he and Ungar published a paper about the architectural benefits of mirrors [BU04].

After the project moved to Sun (in 1991) and Smith rejoined us, he pointed out that the benefits of mirrors came at the cost of uniformity. In thinking about models for systems, Smith always turned to the physical world, which does not support a distinction between direct and reflective operations. There is no difference between a physical object used directly or reflectively: it is the same physical object either way. Furthermore, Smith noted that there are many different types of reflective operations, and any attempt to distinguish between reflective and non-reflective operations was therefore certain to get it wrong in some cases or from some points of view. In fact, we sometimes do find it unclear whether a method should take a mirror as argument or the object itself.

Smalltalk, in contrast, placed many reflective operations in the root of the inheritance hierarchy, to provide reflection for every object. Something similar might have been done for Self, to avoid the dichotomy that was bothering Smith. However, by this point we were reveling in Self's support for lightweight objects that needed no place in the inheritance hierarchy, and it would have been impossible to reflect upon such objects without mirrors. Ungar still feels that reflective operations are of a different breed, while Smith still wishes they could be unified with ordinary operations.

Another approach to unifying invocation and access would have been to add a bit to a slot to record whether the slot was a method or data slot. Then, a special operation could have extracted a method from a method slot and put it into a data slot. This approach would have had its own problems: what would it mean to put 17 into a method slot? In our opinion, we never fully resolved whether a method should fire because it is a method or because it is in a special kind of slot. We also continue to wrestle with writing code in which it is unclear whether we should pass around objects or mirrors on them. There was also the efficiency loss in creating an extra object to do reflection. The performance penalty went unnoticed when we were using mirrors for a programming environment, but became problematic in the Klein system [USA05], which mirrors to convert a hundred thousand objects to a different representation. In this application the efficiency loss was so critical that a switch

9. This operation was inspired by, but not quite the same as, Smalltalk's "become:" operation.

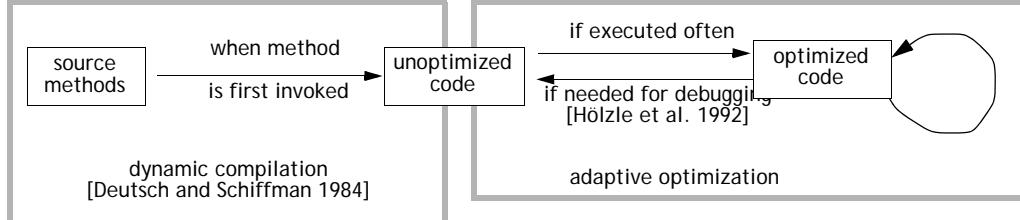


Figure 14. Compilation in the Self-93 system.

was added to the virtual machine to support a model in which a method could be stored and retrieved from a variable slot. Although Ungar feels that the preponderance of evidence weighs on the mirror and methods-firing-by-themselves side of the debate, Smith acknowledges clear advantages for mirrors but still harbors doubts, and this issue is not completely resolved.

Recall that, when we first saw Self objects on the screen at Stanford in 1988-1989, we realized that some objects had too many slots to view all at once: we needed some way to subdivide them, much like Smalltalk’s method categories. To support this non-semantic grouping, *annotations* were added to the language. Via reflection, the system could annotate any object or any slot with a reference to an object. The virtual machine supported the annotation facility but ignored the annotation contents. It optimized the space required when all objects cloned from the same prototype contained the same annotations by actually storing the annotations in the maps. The programming environment (written in Self) used the annotations to organize an object’s slots in (potentially nested) categories.

5. Self Moves to Sun and Becomes a Complete Environment

As Ungar and his students worked on mirrors in California, Smith was in Cambridge, England at Rank-Xerox EuroPARC where he had been working on a multi-user version of the Alternate Reality Kit. But his interest in Self and prototype-based languages persisted, and while in Cambridge he was able to work with Alan Borning and Tim O’Shea, who had connections with the PARC Smalltalk group. With these two plus Thomas Green and Moira Minoughan, also working at EuroPARC, he explored a few other language ideas [GBO99].

On his return to Xerox PARC at the end of 1989, Smith was amazed to find Self running so well, though a little concerned that it had acquired complexities such as multiple inheritance, mirrors, and annotations (which he felt were too much like Smalltalk’s method protocols, having no runtime semantics). He decided to join Ungar and carry Self forward into a larger implementation effort. Smith had been thinking about subjectivity in programming languages, but further language work was becoming a harder sell to PARC management. The authors decided to take the Self ideas to other research labs. (We later returned to subjectivity in [SU96].) By the end of June 1990, the Self team had given talks on the developing Self system at U.C. Berkeley, PLDI’90, and IBM Hawthorne Laboratories. In the fall of 1990, we considered moving to the Apple Advanced Technology Group, but—encouraged by Emil Sarpa, Bill Joy, and Wayne Rosing—decided to join Sun Microsystems’ research labs. Self already ran on the SPARC processor and thus there was a chance to get a leg up in adoption. The labs were just being formed, and the Self project would be one of Sun Labs’ first groups. In January 1991, the Self project joined Sun Microsystems Laboratories. Ungar’s students (Craig Chambers, Bay-Wei Chang, Urs Hölzle, and Ole Agesen, the last graduate stu-

dent to join the project) became consultants and over the years more researchers were hired to work on the project: John Maloney, Lars Bak, and Mario Wolczko.

5.1. More Implementation Work

As 1991 ended, the virtual machine encompassed 75,000 lines of code; in 1992, our first Apple laptop computers arrived and we started work on our first Macintosh port. By the end of 1992, Lars Bak had obtained a 5x speedup on the Sun computers for Self’s browsing primitives (implementers, etc.) by rewriting the low-level heap-scanning code. He had also trimmed Self’s memory footprint by 18%. The Macintosh port went slowly at first; it was not until January 1996 that Self ran (with an interpreter) on a PowerPC Macintosh.

Recall from section 4.1 that the third Self compiler ran the benchmarks pretty well but still compiled too slowly, and suffered from brittle performance. Hölzle took up the challenge. He had built polymorphic inline caches (PICs) [HCU91], and then proposed a new direction: laziness. He suggested that we build two compilers: a fast-and-dumb compiler that would also include instrumentation (such as counters in PICs), and a slow-and-smart compiler that would reoptimize time-consuming methods based on the instrumentation results (see Figure 14). The first time a method was run, the system used a fast-but-unsophisticated compiler that inserted an invocation counter in the method’s prologue. As the method ran, its count increased and its call sites became populated with inline caches. Periodically, another thread zeroed out the counters. If a method was called frequently, its counter would overflow and the virtual machine would recompile and optimize it. However, because the method with the overflowing counter might have been called from a loop, the system would walk up the stack to find the root method for recompilation. After selecting the root, that method would be compiled with a slow-but-clever optimizing compiler that would exploit the information in the inline caches to inline callees. Finally, the set of stack frames for the recompiled methods would be replaced by a stack frame for the optimized methods (called “on-stack replacement”), and execution would resume in the middle of the optimized method.¹⁰

By the time Hölzle was through, his system, Self-93, ran well indeed, with almost no pauses for compilation [Hölzle 1994, HU94, HU94a, HU95, HU96]: in a 50-minute interactive graphical session, using a new metric that lumped together successive pauses, we found that, on a 28.5 MIPS, 40 MHz SPARCstation-2, two-thirds of the lumped pauses were less than 100ms, and 97% were less than a second [HU94a]. This system reduced the time to start the graphical user interface from 92 to 26 seconds. Not only were pauses reduced, but benchmarks sped up. This system ran a suite of six large and three medium-sized programs 1.5 times faster than the third-generation Self system.

10. In later years, when Ungar had to maintain and port the virtual machine single-handed, he would disable on-stack replacement to simplify the system and eliminate hard-to-reproduce bugs.

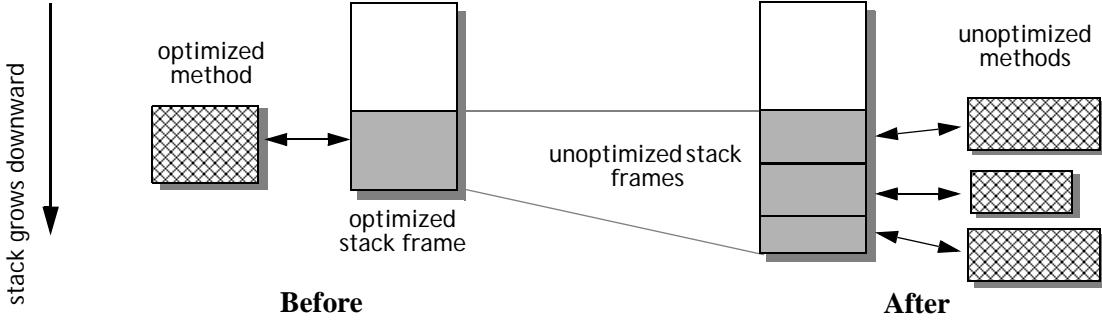


Figure 15. Transforming an optimized stack frame into unoptimized form.

As Hözle was performing this feat of legerdemain, Ungar was worried about preserving the system's transparency. He wanted the system to feel like a fast interpreter, and that meant that if the user changed a method, any subsequent call to the changed method had to reflect the change. However, if the method had been inlined and if execution were suspended in the caller, when execution resumed the calling code would proceed to run the obsolete, inlined version of the callee. To remedy this problem, Ungar suggested on-stack replacement in reverse: replacing the one, optimized stack frame for many methods with multiple stack frames for unoptimized methods (see Figure 15). Hözle brought this idea to life [HCU92], and Self's sophisticated optimizations became invisible to the programmer. We had finally realized our vision for the virtual machine: high performance for Self, a pure and dynamic language, combined with high responsiveness and full source-level debugging. However, Self's virtual machine required many more lines of C++ code (approximately 100,000), more complexity, and more memory than contemporary (but slower) Smalltalk virtual machines.

In August 1993, Mario Wolczko left the University of Manchester where he had been working on a Smalltalk multiprocessor, and joined our project and performed some space engineering, cleaned up the representation of debugging information, refactored the implementation of maps, and fixed a large number of bugs. He also implemented a feedback-mediated control system that managed old-space collection and heap expansion. The policy was implemented in Self, with only the bare bones mechanism in the virtual machine. This work was ahead of its time, and we are unaware of its match in current systems.

Ole Agesen was the last PhD student in the Self project, graduating in 1996. He worked on many portions of the Self system, including an interface to the dynamic linker for calling library routines, and for his dissertation built a system that could infer the types of variables in Self programs, despite Self's lack of type declarations. Agesen's work showed how to prune unused methods and data slots from a Self application [APS93, AU94, Age95, AH95, Age96].

5.2. Cartoon Animation for UI1

With the Alternate Reality Kit, Smith wanted to deliver a feeling of being in a separate world by having lots of independent things happening in a physically realistic and often subtle way. He tried for realistic graphics, including shadows and avoiding outlines, but never even thought about cartoon animation techniques, in which fidelity to physics is less important than emphasizing certain motions through physically implausible accelerations and deformations. In building the UI1, however, the Stanford group believed that a physical feel would be a help to the programmer,

and after seeing ARK, were convinced that animation should feature heavily in any Self user interface.

When he moved Sun in 1991, Ungar had been watching a lot of Road-Runner and Popeye cartoons with his five-year-old son, Leo. It occurred to Ungar that the animation techniques he saw in the cartoons could be applied to dynamic user interfaces. Since he also had a VCR with an exceptionally agile jog-shuttle feature, he was able to review many scenes one frame at a time. Smith and neuroscientist Chuck Clanton (who was then consulting at Sun) were also fascinated by animation.

Coincidentally, in 1990 Steven Spielberg and Warner Brothers put *Tiny Toon Adventures* on the air, a show that strove to recreate the style and quality of the classic Warner Brothers cartoons in the late 1930s through early 1950s. In our first year at Sun, Smith and Ungar would stop work every day at 4:30 to watch these cartoons and then dissect them. The cartoons inspired us to read Thomas and Johnson's book *Disney Animation: The Illusion of Life* [TJ84] and *Road-Runner*, director Chuck Jones' autobiography [Jone89]. We would stare in fascination at each frame of the Road-Runner zooming across the screen. We were struck by the clarity with which Jones could show a scrawny bird and an emaciated coyote crossing the entire width of a movie screen in only a handful of frames by using motion blur and slowly dissipating clouds of dust. This combination of speed and legibility stood in stark contrast to the leisurely pace of many animated computer interfaces of the time in which each small change of position was painstakingly redrawn. Even some of the best interface research at the time used uniform, unblurred motion [RMC91]. We started thinking about the role motion blur could play in graphical computer interfaces. Smith asked the key question: "If you could update the screen a thousand or a million times a second, would you still need motion blur?" These explorations led us to an understanding of how to bridge the gap between cartoons and interfaces, and how to make changes more legible without slowing things down.

There is an interesting difference between Smith's use of animation in the Alternate Reality Kit and cartoon animation. In any animation, the incoming light creates patterns on the viewer's retina that trickle up the nervous system and reach the higher levels of cognition after considerable processing. In ARK, Smith's goal was to create a sense of realism by replicating the retinal patterns caused by real-world objects. In contrast, cartoon animation is more concerned with getting the viewer's higher cognitive levels to perceive objects and motion. In depicting a bouncing billiard ball, a cartoonist might use a "squash and stretch" around the moment of impact, thereby making the bounce clearly legible to viewers. A literal moment by moment capture of the human retina watching an actual billiard ball bounce would reveal a blur that only somewhat resembles cartoon-style stretch. Although both kinds of animation

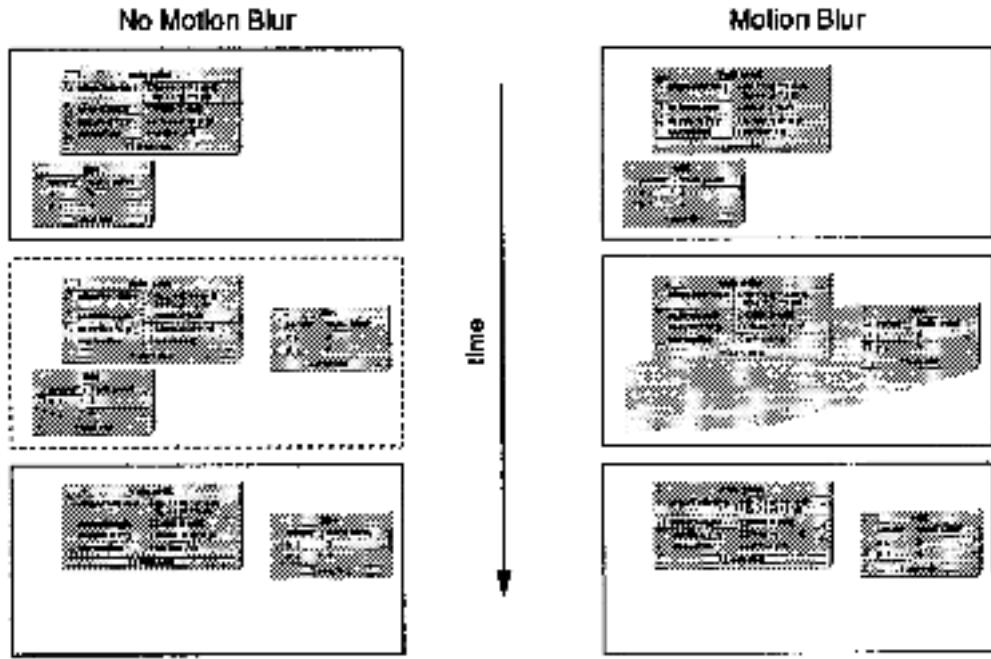


Figure 16. When objects are moved suddenly from one position to another, it can seem as if there are two instances of it on the screen at the same time. The eye sees something like the middle frame of the “no-motion-blur” figure, even though such a frame doesn’t actually ever appear on the screen. Motion blur reduces this effect and gives a visual indication of the object’s travel, so that it is easy to see which object moved where.

have the same goal, cartooning trades the literal replication of sensory inputs for better legibility at higher levels of cognition.

Recall that colormap trickery enabled UI1 to display 20 to 30 frame-per-second smooth-looking animation on 1991 hardware (section 4.3). Once the basic techniques were implemented, we set about using cartoon animation everywhere we could to improve UI1’s feel and legibility. (A popular phrase at the time was: “moving the cognitive burden from the user to the computer.”) In those days, windows, menus, and dialog boxes would just appear in a single frame and vanish just as abruptly. (As of 2006, they still do in many window systems.) But, we believed that every abrupt change startled the user, and forced him or her to involuntarily shift his or her gaze. So, we strove to avoid bombarding the user with abruptly changing pixels. Just as the Road Runner would enter the frame from some edge, every new Self object appearing on the screen would drop in from the top, slowing down as it did, and wiggling for an instant as it stopped. Every pop-up menu would smoothly zoom out, then the text would fade in. We became excited about the user experience that was emerging. Ungar came in every day over one Christmas break (probably 1991) to get good-looking motion blur into the system.

Figure 16, taken from [CU93], illustrates motion blur. Chang realized that objects should move in arcs, not straight lines, and also suggested that an object wiggle when hit by a sprouted arrow (see Figures 17 through 19). Ungar played with the algorithm and its parameters until he got the wiggle to look just right. It would have been much more difficult to break this new ground with any other system: he needed both Self’s instant turnaround time to try ideas freely, and also its dynamic optimizations so that the animation code would run fast enough.

Table 1 summarizes UI1’s cartoon animation techniques. By June 1992, we had implemented all of our cartoon animation, including motion blur, menu animation, and contrast-enhancing highlighting of menu selection. Chang had also started video taping users to evaluate UI1, taping eight subjects before completing his dissertation. His work on cartoon animation and its effect on users’ productivity became a part of his dissertation and was presented in several conferences [CU93, CUS95, Chan95]. Although Ungar also wanted to measure the effect of animation on the number of smiles on users’ faces, we never did. Now, in 2006, many of these techniques can be seen in commercial systems and web sites.

5.3. UI2 and the Morphic Framework

When Smith rejoined the group on the move to Sun, he was thinking of ways to push the UI1 framework in additional directions. He felt the analogy to physical objects was not taken far enough in most user interfaces. For programming purposes, he felt that the analogy meant that every object should be able to be taken apart, even as it is running. This physicality was after all the goal of the language-level objects, and with a tight correspondence between on-screen object and language-level object, the deconstruction of live on-screen objects seemed to complete the paradigm.

In working with physical objects, one is free to take them apart and rearrange parts even while the universe continues to run: there is no need to jump to a special set of tools in a different universe. Physical objects do not support a use/mention distinction: the hammer in use is the same as the hammer examined for improvement, repair, or other modifications. So Smith wanted to be able to pick up a scroll bar from a running word processor and reattach it to some other application.



Figure 17. On a click, a menu button transforms itself from a button into the full menu. After a selection has been made, it shrinks back down to a button. (In this and other figures, only a few frames of the actual animation are shown. These figures taken from [CU93].)

The UI1 interface contained only representations of objects: there was no special support for building conventional GUIs with elements such as scroll bars, text fields, buttons and sliders. Mainly for this reason, Smith started an effort within the group to build UI2, a new framework that would retain the basic object representation idea and animation techniques of UI1, but add the ability to create general GUIs [MS95, SMU95]. In keeping with the language concepts of malleability and concreteness, within UI2 it would be possible to take objects apart directly: thus a direct copy-deconstruct-reconstruct method would be an important part of building new GUIs based on recognizable GUI widgets.

John Maloney, hired at this time, began creating much of the UI2 framework, which came to be called the Morphic framework. The Morphic framework enhanced the sense of direct

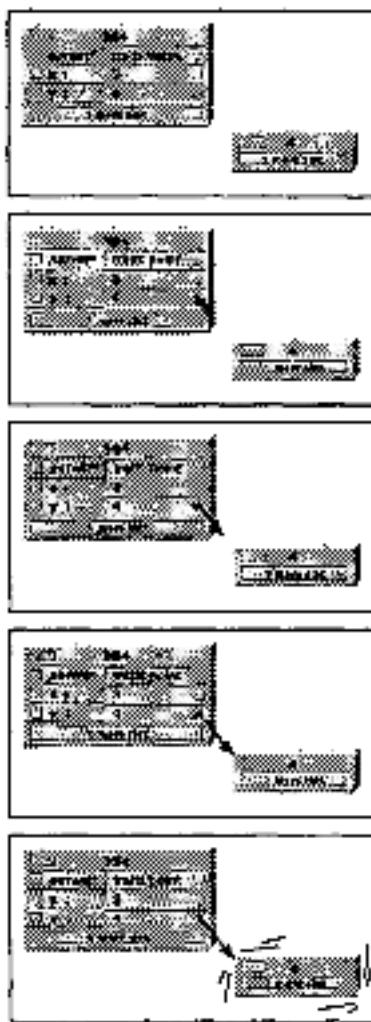


Figure 18. Arrows grow from their tail to hit their target. The target reacts to the contact with a small wiggling jolt (here suggested by a few lines). Arrows also shrink back down into their tail.

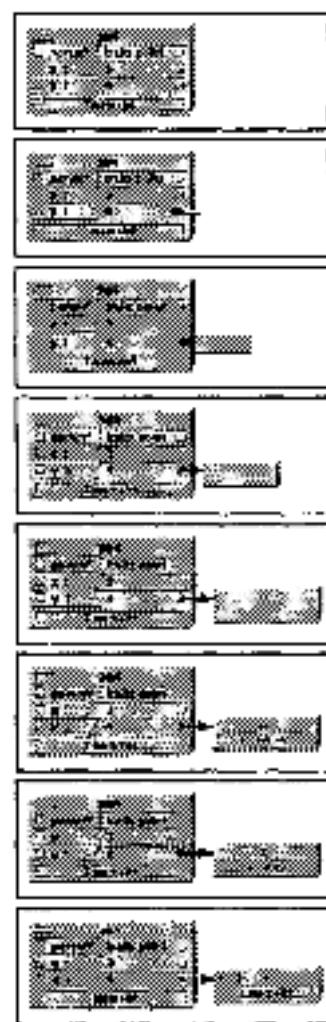


Figure 19. Objects grow from a point to the full-size object; any connecting arrow grows smoothly along with the object. Currently, text does not grow along with the object, instead fading in smoothly on the fully grown object.

manipulation by employing two principles we called *structural reification* and *live editing*.

Structural Reification. We decided to call the fundamental kind of display object in UI2 a “morph,” a Greek root meaning essentially “physical form.” Self provides a hierarchy of morphs. The root of the hierarchy is embodied in the prototypical morph, a kind of golden-colored rectangle. Other object systems might choose to make the root of the graphical hierarchy an abstract class with no instances, but prototype systems usually provide generic examples of abstractions. This is an important part of the structural reification principle: there are no invisible display objects. The root morph and its descendants are guaranteed to be fully functional graphical entities. Any morph inherits methods for displaying and responding to input events that enable it to be directly manipulated.

Table 1: Summary of UI1 Cartoon Animation Techniques (from [CU93])

Technique	Principle	Examples from Cartoons	Examples from the Self Interface
Solidity	solid drawing	<ul style="list-style-type: none"> Parts of Snow White's dwarves may squash and stretch, but they always maintain their connectedness and weight 	<ul style="list-style-type: none"> Objects move solidly Objects enter screen by traveling from off screen or growing from a point Menus transform smoothly from a button to an open menu Arrows grow and shrink smoothly Transfer of momentum as objects respond to being hit by an arrow
	motion blur	<ul style="list-style-type: none"> Road Runner is a blue and red streak 	<ul style="list-style-type: none"> Stippled region connects old and new locations of a moving object
	dissolves	<ul style="list-style-type: none"> n/a 	<ul style="list-style-type: none"> Objects dissolve through one another when changing layering
Exaggeration	anticipation	<ul style="list-style-type: none"> Coyote rears back onto back leg before chasing after Road Runner 	<ul style="list-style-type: none"> Objects preface forward movement with small, quick contrary movement
	follow through	<ul style="list-style-type: none"> Road Runner vibrates for an instant after a quick stop 	<ul style="list-style-type: none"> Objects come to a stop and vibrate into place Objects wiggle when hit by an arrow
Reinforcement	slow in and slow out	<ul style="list-style-type: none"> Coyote springs up from ground, with fastest movement at center of the arc 	<ul style="list-style-type: none"> Objects move with slow in and slow out Objects and arrows grow and shrink with slow in and slow out Objects dissolve through other object with slow in and slow out Text fades in onto an object with slow in and slow out
	arcs		<ul style="list-style-type: none"> Objects travel along gentle curves when they are moving non-interactively
	follow through		<ul style="list-style-type: none"> Objects do not come to a sudden standstill, but vibrate at end of motion

In keeping with the principle of structural reification, any morph can have “submorphs” attached to it. A submorph acts as if it were glued to the surface of its hosting morph. Composite graphical structure typical of direct-manipulation interfaces arises through the morph-submorph hierarchy. Many systems implement composition using special “group” objects, which are normally invisible. But because we wanted things to feel very solid and direct, we chose to follow a simple metaphor of sticking morphs together as if with glue.

A final part of structural reification arose from our approach to laying out submorphs. Graphical user interfaces often require subparts to be lined up in a column or row: Self’s graphical elements are organized in space by “layout” morphs that force their submorphs to line up as rows or columns. John Maloney was able to create efficient “row and column morphs” as children of the generic morph that were first-class, tangible elements in the interface. A row morph holding four buttons aligned in a row is at the bottom of the ideal gas simulation frame in Figure 22. Row or column morphs embody their layout policy as visible parts of the submorph hierarchy, so the user need only access the submorphs in a structure to inspect or change the layout in some way. The user who did not care about the layout mechanism paid a price for this uniformity, and was confronted with it anyway while diving into the visual on-screen structures.

Live Editing. Live editing simply means that at any time the user can change any object by manipulating it directly. Any interactive system that allows arbitrary runtime changes to its objects has some support for live editing, but we wanted to push that to apply to the user interface objects directly. The key to live editing is UI2’s “meta menu,” a menu that pops up when the user holds down the third mouse button while pointing to a morph. The meta menu contains items such as “resize,” “dismiss,” and “change color” that let the user edit the object directly. Other menu elements enable the user to “embed” the morph into the submorph structure of a morph behind it, and give access to the submorph hierarchy at any point on the screen.

Lars Bak did a lot of work to create the central tool for programming within UI2, the object “outliner,” analogous to the Smalltalk object inspector. (We were in part inspired by “MORE,” an outlining program we had recently started using.) The outliner shows all of the slots in an object and provides a full set of editing facilities. With an outliner you can add or remove slots, rename them, or edit their contents. Code for a method in a slot can be edited. Access to the outliner through the meta menu makes it possible to investigate the language-level object behind any graphical object on the screen. The outliner supports the live-editing principle by letting the user manipulate and edit

slots, even while an object is “in use.” Figure 20 shows an outliner being fetched onto the screen for an ideal gas simulation.

Recall that popping up the meta menu is the prototypical morph’s response to clicking the third mouse button. All morphs inherited this behavior, even elements of the interface like outliners and pop-up menus themselves. But our Self pop-up menus were impossible to click on: one found them under the mouse only when a mouse button was already down. Releasing the button in preparation for the third button click caused a frustrating disappearance of the pop-up. Consequently, we provided a “pin down” button that, when selected, caused the menu to become a normal, more permanent display object. The mechanism was not new, but providing it in Self enabled the menu to be interactively pulled apart or otherwise modified by the user or programmer. It is interesting to compare this instance of the use-mention problem and solution with the analogous use-mention issue that faced us at the language level: that of accessing a method object in a slot without running the method. There, the solution was to introduce a special mentioner object, the mirror (see section 4.4).

Live editing is partly a result of having an interactive system, but it is enhanced by user interface features that reinforce the feel that the programmer is working directly with concrete objects. The example running through the rest of this section will clarify how this principle and the structural reification principle help give the programmer a feeling of working in a uniform world of accessible, tangible objects.

Suppose the programmer (or motivated user) wishes to improve an ideal gas simulation by extending the functionality and adding user interface controls. The simulation starts simply as a box containing “atoms” that bounce around inside. Using the third mouse button, the user invokes the meta menu to select “outliner” to get the Self-level representation of the object (Figure 20). The outliner makes possible arbitrary language-level changes to the ideal gas simulation.

Now the user can start to create some controls right away. The outliner has slots labeled “start” and “stop” that can be converted into user interface buttons by selecting from the middle-mouse-button pop-up menu on the slot. Pressing these buttons starts and stops the bouncing atoms in the simulation. In just a few gestures the user has gone through the outliner to create interface elements while the simulation continues to run.

The uniformity of having “morphs all the way down” further reinforces the feel of working with concrete objects. For example, the user might wish to replace a textual label with an icon. The user begins by pointing to the label and invoking the meta menu. The menu item labeled “submorphs” lets the user select which morph in the collection under the mouse to denote (see Figure 21). The user can remove the label directly from the button’s surface. In a similar way, the user can select one of the atoms in the gas tank and duplicate it; the new atom can serve as the icon replacing the textual label. Structural reification is at play here, making display objects accessible for direct and immediate modification.

As mentioned above, all the elements of the interface such as pop-up menus and dialog boxes are available for reuse. Say the user wants the gas tank in the simulation to be “resizable” by the simulation user. The user can create a resize button for the gas tank simply by “pinning down” the meta menu and removing the resize button from it. This button could then be embedded into the row of controls along with the other buttons (see Figure 22).

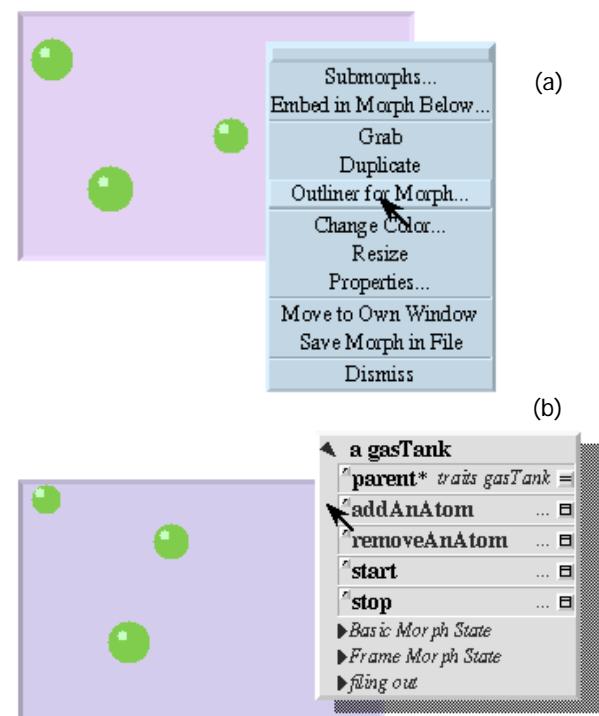


Figure 20. In UI2 the user pops up the meta menu on the ideal gas simulation (a). Selecting “outliner” displays the Self-level representation, which can be carried and placed as needed (b). (The italic items at the bottom of the outliner are slot categories that may be expanded to view the slots. Unlike slots, categories have no language level semantics and are essentially a user interface convenience.)

During this whole process, the simulation can be left running: there is no need to enter an “edit” mode or even to stop the atoms from bouncing around. The live editing principle makes the system feel responsive, and is reminiscent of the physical world’s concrete presence.

5.4. From UI2 to Kansas

Smith was fascinated by shared spaces (we might now call them “shared virtual realities”) and had explored with a shared version of his Alternate Reality Kit during his year at Rank-Xerox EuroPARC (1988-1989) [GSO91, SOSL97, Setal93, Setal90, Smi91]. After he and Ungar joined Sun and the UI2 framework was underway, Smith decided to make UI2 into a shared world in which the team could work together simultaneously. The transformation took only a day or two of diligent work, thanks in part to the fact that the system was built on top of the X windowing system, and of course in part to the fact that Self was intended to be a flexible system allowing deep, system-wide changes. The idea was to transform a single Self world with a single display into a single Self world with potentially many displays that could be anywhere on the network. Thus, whenever a morph displayed itself, rather than merely use the local X display window, the code would iterate over a list of several display windows, some of them remote. Moreover, the entire list of windows was queried for events to be dispatched to appropriate objects in the central Self world.

In addition, each remote window had an associated offset so that, although several users could be in the space at once, they could be shifted to individual locations. Because the resulting

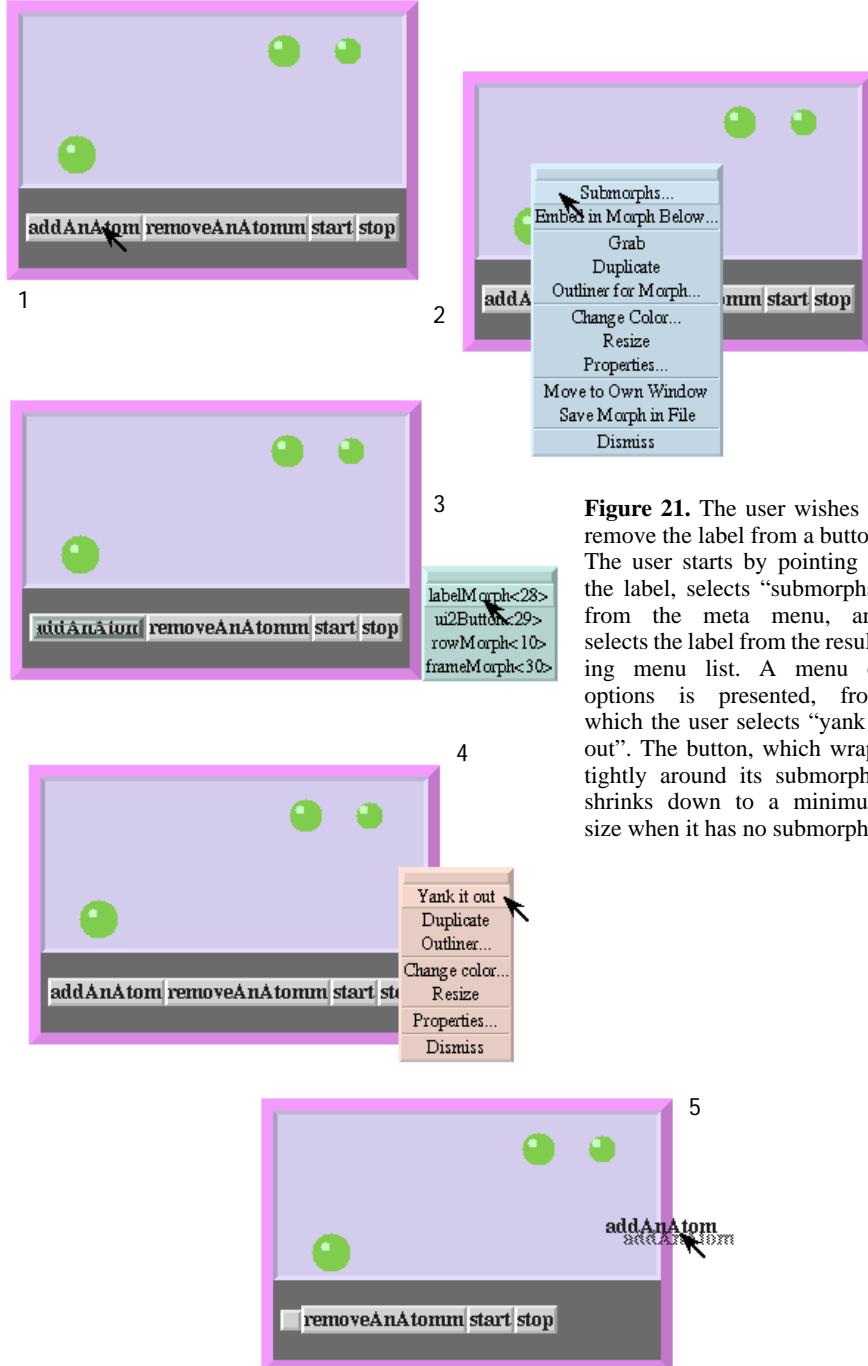


Figure 21. The user wishes to remove the label from a button. The user starts by pointing to the label, selects “submorphs” from the meta menu, and selects the label from the resulting menu list. A menu of options is presented, from which the user selects “yank it out”. The button, which wraps tightly around its submorphs, shrinks down to a minimum size when it has no submorphs.

effect was to put many people in a single, vast flat space, we named the system “Kansas.”

When Kansas was running, an error in the display code could cause all the windows to hang: the other threads of the underlying Self virtual machine would continue to run, though the display thread was suspended. We decided the right thing to do here was to open up a new shared space, “Oz,” that would be created by the same virtual machine that created Kansas, but would be a new (mainly empty) world containing only a debugger on the suspended thread. The users, finding themselves suddenly “sucked up” into the new overlaying world of Oz, could

collaboratively debug and fix the problem in Kansas, then, as a final act in Oz, resume the suspended thread so that normal Kansas life might resume. Much of the work for this was done by Mario Wolczko; Smith, Wolczko, and Ungar wrote a description for a special issue of the *Communications of the ACM* on debugging [SWU97].

UII was beautiful; its use of cartoon animation techniques gave a smooth and legible appearance. However, it gave no help to the user who wanted to either dissect or create a graphical object. We wanted to remedy this shortcoming in Morphic, and replicating UII’s beauty took a back seat to architectural innova-

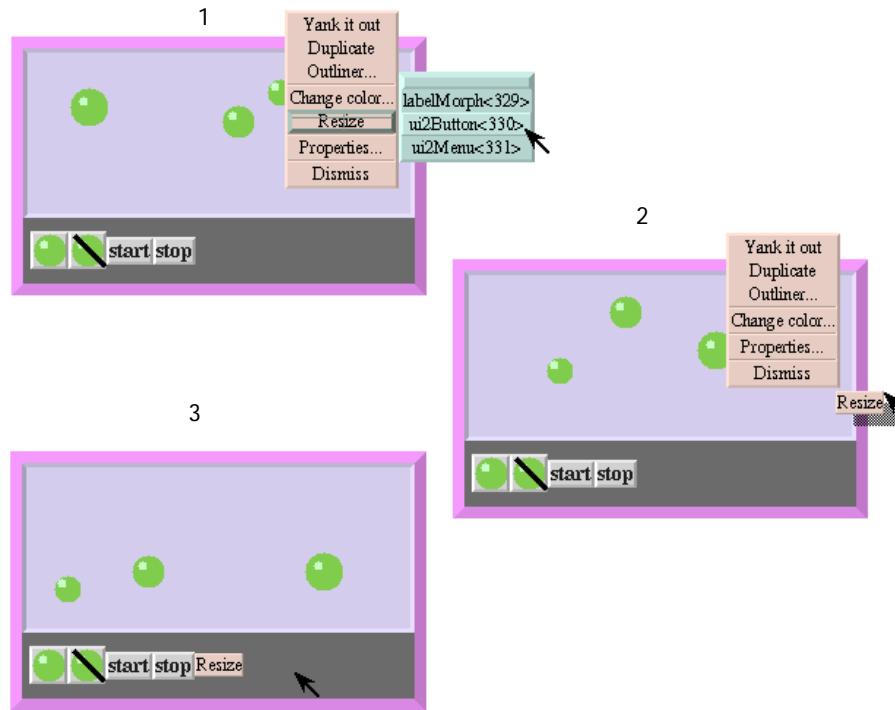


Figure 22. The environment itself is available for reuse. Here the user has created the menu of operations for the gas tank, which is now a submorph of the surrounding frame. The user has “pinned down” this menu by pressing the button at the top of the menu, and can then take the menu apart into constituent buttons: here the user chooses the resize button for incorporation into the simulation.

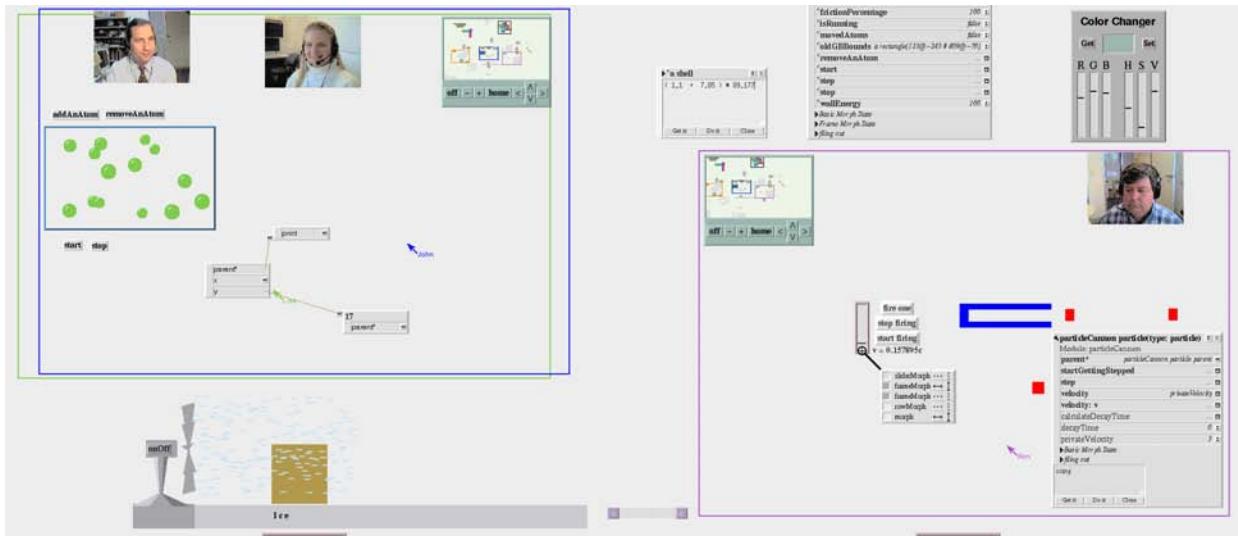


Figure 23. The Kansas shared space version of UI2. Here three users are shown, two of whose screens largely overlap so they can see each other and work on a common set of objects; the third user to the right is by himself. Video images from desktop cameras are sent over the network to appear on special objects near the top of each user’s screen boundary. Users can be aware that other users are nearby thanks to audio from each user that diminishes in volume with distance, and to the miniature “radar view” tools that give an overview of the nearby extended space (a radar view can be seen in the upper left corner of the rightmost user’s screen). The radar view can be used to navigate through the larger space as well.

tion. With the exception of slow-in and slow-out, cartoon animation techniques such as smooth dissolves, motion blur, anticipation, follow-through, and “squash and stretch” were never incorporated into the Morphic framework.

After the Self project ended in 1995, Smith became interested in distance learning. By 1998, he had added desktop video and desktop audio to the system, as depicted in Figure 23. In 1998 and 1999, Smith used this audio- and video-link-enabled Kansas in a series of experiments comparing small co-present study groups with small study groups in Kansas who communicated using the audio and video links. In these studies, groups of five to seven students in the same course would not attend lectures at all, but rather gather (either in the same room or in Kansas) to discuss a video tape of the classroom lectures. The experiments were carried out at California State University at Chico, approximately 200 miles north of the Sun Labs campus. A special Kansas world of Self objects was installed in a network at Chico, and the Sun researchers could “drop in” remotely, and even reprogram the system while a study session was in progress. The results of these experiments were that there were no significant difference between student grades, although there were some behavioral differences [SSP99, Setal99].

The audio- and video-link technologies were specific to Sun workstations; since we wanted Self to run on a wider set of platforms, they were never part of the mainstream Self system. But users of Self today can share their screens to work together. In normal use, Self programmers seem to prefer having their own private world of Self objects, which discourages routine use of the Kansas features, though they are still often useful for remote demos and collaborative debugging or development sessions. (In fact, Ungar has been recently using it to work with a collaborator 3,000 miles away.)

5.5. The Transporter

Inspired by Smalltalk and then ARK, we wanted to submerge the Self programmer in a world of live objects as opposed to some text editor. In fact, from the start of Self in 1986, we hoped that Self’s prototypes would feel even more alive than Smalltalk’s classes. Once we had a decent virtual machine and UI1, we could experiment with this idea: we could create objects, interactively add slots and methods, try them out, and instantly change them.

Although the idea of programming in a sea of live objects was inspired by Smalltalk, Self was not Smalltalk, and the differences caused problems. In Smalltalk, the programmer creates classes, instantiates them, changes methods, and inspects objects. A Smalltalk method is always part of a class, a class is a special kind of object, and every class (by convention) resides in the same spot: the System Dictionary. So, a Smalltalk program can usually be considered a collection of class definitions, including any methods. Since a Smalltalk object is created by instantiation, all initialization had to be done programmatically, and initialization is not much of a special problem. But in Self, there are no “class” objects. There is no one System Dictionary. Any object may serve as a namespace, and any object may hold methods. Objects are typically created by copying prototypes, and initialization code is frowned upon, as the prototype is already supposed to be initialized, functional, and prototypical. In fact, the prototype ideal (as Smith used to say) is to always have everything initialized so that every prototype can be functional as is. But this view means that the state of objects is an integral part of a “program.” Consider the following example: Suppose Alice, using her own object heap, writes a program that she wishes to give to Bob. Bob will typically be using his own

object heap, and so needs to incorporate Alice’s additions and changes. In Smalltalk, the additions and changes that were typically a part of a program were restricted, but in Self, the problem amounted to recreating arbitrary changes to objects.

Up to around 1992, whenever we wanted to “get serious” so as to share our work with the group, we wrote Self programs in a text editor, then read them in and debugged them. As we debugged, we had to either change the file and reread it, or change both the file and the running environment. This was painful. We needed a system that would turn arbitrary sets of objects and slots into a text file that could then be read in to another world of objects. To meet this need, Ungar started his last major effort in the Self project, the transporter [Ung95].

Ungar realized that Self “programs” involved adding slots to or modifying slots in existing objects and thus the transporter would have to operate at the level of individual slots. Slot a might be part of one “program” and slot b another, even though both were in the same object. Since extra information was needed that was not part of the execution model, that information was added to the system around 1994 by extending Self’s existing annotations. Each slot was annotated with the name of the source file to which it would be written. At first, Ungar tried to write a system that would infer other data, such as the proper initialization for a slot when it was subsequently read in. After many unsuccessful attempts, it became clear that inference would not work, and more information would be needed in the annotations. For example, each slot had to be annotated with initialization instructions: should it be initialized to whatever it originally contained or to the results of some expression? At the end, Ungar came to a fundamental realization: what was later dubbed “orthogonal persistence” [AM95] was, at least in this context, a flawed concept. Simply making a set of slots persistent is easy. But installing those slots into another arbitrary world of objects so that they function as they did in their original home is difficult, and probably even impossible in the general case. The task at least requires more information than what is needed for the slots merely to function in a live image. The programmer has to provide information that will enable the slots to function as the programmer intends (see Figure 24). To keep the burden of specifying the extra information for the transporter as light as possible, Ungar integrated it into the programming environment in such a way that it would be easy for a programmer to “push a button” and save a program as a source file that could then be read in to another user’s heap of objects.

In June of 1994 the transporter was finished and the programming environment was augmented with affordances for the extra information. The Self team had moved its 40K lines of Self code, comprising data and control structures, the user interface, and the programming environment, to the Self transporter; in other words, we were all (but Agesen) doing our Self programming inside the graphical programming environment. The Self team made a leap from programming in text editors to programming in a live world, and then transporting the results to text files for sharing with others.

5.6. Significant Events While at Sun

The first Self release, 1.0, had occurred at the end of September 1990 and went to over 100 sites. The next release, 1.1, came at the end of June 1991 and went to over 150 sites. It featured a choice of compilers, and support for lightweight processes. Released in August 1992, Self 2.0 featured the sender-path tie-breaker and shared-part privacy rules. It also introduced full source-level debugging of optimized code, adaptive optimization to shorten compile pauses, lightweight threads within Self,

<code>compilerProfiling</code>	<code>compilerProfiling =</code>
<code>enumerating</code>	<code>enumerating =</code>
<code>false</code>	<code>false =</code>
<code>fileTable</code>	<code>fileTable = <i>a vector<565>(64 elements)</i></code>
<code> Module unix</code>	
<code> Follow</code>	<code>Initialize to <i>vector copySize: 64</i></code>
<code>flatProfiling</code>	<code>flatProfiling =</code>
<code>historyListEntry</code>	<code>historyListEntry[1] =</code>
<code>host</code>	<code>host =</code>
<code>infinity</code>	<code>infinity = inf</code>
<code> Module float</code>	
<code> Follow</code>	<code>Initialize to</code>
<code>interceptor</code>	<code>interceptor =</code>
<code>maxSmallInt</code>	<code>maxSmallInt = 536870911</code>

Figure 24. Annotations for transporting slots. The ovoid outlines have been added to show the module and initialization information for two slots (named `fileTable` and `infinity`) in a Self object.

support for dynamically linking foreign functions, support for changing programs within Self, and the ability to run the experimental Self graphical browser under OpenWindows. Self Release 2.0 ran on Sun-3s and Sun-4s, but no longer had an optimizing compiler for the Motorola 68000-based Sun-3 (and therefore ran slower on the Sun-3 than previous releases). Self 3.0, featuring Hözle's adaptive optimization and the simplified multiple inheritance and privacy rules, was released January 1, 1994. In 1995, we felt we had achieved a fully self-contained system: we had an elegant language with a clever implementation unified with a novel user interface construction / programming world. Having made the final determination of which features to implement for the release by voting with hard candy (see Figure 25), we released Self 4.0 into the larger world as a beta in February 1995, and in final form July 1995. (See appendix 11 for the actual release announcements.)

During this 1991-1995 period at Sun, we felt that Self was gaining a following, especially within the academic community. Craig Chambers and Ungar gave a tutorial at the 1992 OOPSLA conference describing the various Self implementation techniques that sold out. At the 1993 OOPSLA conference, a demo of the Morphic framework and Self programming environment proved so popular that we had to schedule a second, then a third showing due to overflowing crowds. That same year, we presented the project to Sun CEO Scott McNealy, who enthused about Self being "a spreadsheet for objects" but cautioned us about "fighting a two-front war."¹¹

At one of the OOPSLA conferences in the late 1980's, Ungar had met Ole Lehrmann Madsen. Madsen, a former student of Kristen Nygaard, was one of the designers of the Beta language [MMN93], a professor at Århus University and the advisor of Ole Agesen, Lars Bak, and Erik Ernst. Ungar recalls Madsen proudly explaining how Beta supported virtual classes, and Ungar pointing out that the same idiom just fell out of Self's semantics with no special support required. This may have been

the moment that kindled Madsen's interest in Self. He later sent us Agesen, Bak, and (intern) Ernst, and also spent a sabbatical year with the project in 1994-1995. During his stay, he built a structured editor for Self, and we all enjoyed many rewarding discussions about the various approaches to object-oriented programming. In 1995, Madsen invited the authors to present a paper at the ECOOP conference, giving an overview of the system, emphasizing the common motivational design threads running through Self's language semantics, virtual machine, and user interface [SU95].

We felt that Self might make a good medium for teaching and learning about object-oriented programming, and in 1994-1995 we sponsored work with Brown University and the University of Manchester to develop courses based on Self. In addition to the paper publications, we felt that a videotape might be a good way to present the Self story and in October of 1996 released *Self: the Video*, a 21-minute tape describing the language semantics and shows the user interface, including its shared space aspect [StV96].

Also in 1996, Self alumnus Ole Agesen, who was working at Sun Laboratories, built a system called Pep that ran Java atop the Self virtual machine. It seemed, for a time, to be the world's fastest Java system, demonstrating the potential of Self's implementation techniques for Java programs with a high frequency of message sends [Age97].

When the Self project had joined Sun back in January of 1991, we told the company that we expected to build a fully functional programming environment in three to five years, with six to eight people. Our manager, Jim Mitchell, had us draw up a detailed project schedule. Three years later, we had delivered a fully functional programming environment, user interface, graphical construction kit, and virtual machine. Then, in September 1994, the project was cancelled, possibly in part as a consequence of the company's decision to go with Java. Self was officially wrapped up by July 1995.

11. McNealy never explained what he meant about the two fronts. We suspect he was thinking about asking users to learn both a new language and a new user interface.

Ungar remained at Sun through the summer of 2006 and, aided by Michael Abd-El-Malek and Adam Spitz, kept the Self system alive, including ports to first the PowerPC and then the Intel x86



Figure 25. The Self group decides which features to implement in Self 4.0 by voting with candy (late 1994). Each member distributed a pound of candy among various containers according to which features he desired most. We weighed the results and ate the winners (and the losers, too). Left to right: Randall Smith, Robert Duvall (student intern), Bay-Wei Chang, Lars Bak, John Maloney (seated), Ole Lehrmann Madsen (visiting professor), Urs Hözle, Mario Wolczko, and David Ungar. Not shown: Craig Chambers (who had graduated) and Ole Agesen.

Macintoshes. It remains his vehicle of choice for condensing ideas into artifacts. As of this writing, Self 4.3 is available from <http://research.sun.com/self>.

6. Impact of the Self Project

6.1. The Language

We have always been enthusiastic about the cognitive benefits of unifying state and behavior and of working with prototypes instead of classes. At the 2006 OOPSLA conference, the original Self paper [US87] received an award for being among the three most influential papers published in the conference's first 11 years (from 1985 through 1996). But, for most of the twenty years since Self's design, it was discouraging to see the lack of adoption of our ideas. However, with help from our reviewers (Kathleen Fisher in particular) and from Google (which owes some of its success to Hözle, a Self alumnus), we delightedly discovered that some researchers and engineers working on portable digital assistants (PDAs), programming language research, scripting languages, programming language theory, and automatic program refactoring had been inspired by the linguistic aspects of Self.

Before continuing, we must express our gratitude for all who have put up informative web sites about prototype-based languages, including Ranier Blome [Blom] and Group F [GroF].

6.1.1. Programming Language Research

After he graduated and left the Self project in 1991, Craig Chambers took a faculty position at the University of Washington, Seattle, where he created **Cecil**, a purely object-oriented language intended to support rapid construction of high-quality, extensible software [Cham92a, Cham]. Cecil combined multi-

methods with a classless object model and optional static type checking. As in Self, Cecil's instance variables were accessed solely through messages, allowing instance variables to be replaced or overridden by methods and vice versa. Cecil's predicate objects mechanism allowed an object to be classified automatically based on its run-time (mutable) state. Cecil's static type system relied on types that specify the operations that an object must support, while its dynamic dispatch system was based on runtime inheritance links. For example, any object copied from a "Set" prototype would inherit implementations of "Set" operations such as union and intersection. But there might also be a "Set" type, which promises that any object known at compile time to be that type will implement union and intersection. In private conversations, Chambers has reported to Ungar that his students often struggled with the distinction: when to say "Set (type)" vs. "Set (prototype)." In Ungar's opinion, this illustration of the too-often-overlooked tension between a type system's expressiveness and its comprehensibility is an important result.

The designer of **Omega** [Blas91] wanted to have an object model similar to Self's but did not want to lose the benefits of static type checking. This language managed to unite the two, a surprising feat at the time.

The Self language even influenced a researcher who was deeply embedded in another object-oriented culture, the Scandinavian **Beta** language. Beta is a lineal descendant of Simula, the very first object-oriented language, that features simple, unified semantics based on a generalization of classes and methods called patterns. Like classes, patterns function as templates and must be instantiated before use. Beta allows a designer to model a physical system and then just execute the model, a Beta program [MMM90, MMN93]. In Ungar's opinion, it is one of the cleanest and most unfairly overlooked object-oriented program-

ming languages. Erik Ernst, then a graduate student in the Beta group, spent part of a year interning with the Self project at Sun Labs. After his return to Denmark, he invented gbeta, a superset of Beta that, while still very much in the Beta spirit, added features inspired by Self such as object metamorphosis and dynamic inheritance that straddle the gap between compile- and run-time [Ernst]:

In gbeta, object metamorphosis coexists with strict, static type-checking: It is possible to take an existing object and modify its structure until it is an instance of a given class, which is possibly only known or even constructed at run-time. Still, the static analysis ensures that message-not-understood errors can never occur at run-time...Like BETA, gbeta supports inheritance hierarchies not only for classes but also for methods. This can be used together with dynamic inheritance to build dynamic control structures... [Erns99].

The authors designed but never built **US**, a language that incorporated subjectivity into Self's computational model [SU94, SU96]. An US message-send consisted of several implicit receivers and dispatched on the Cartesian product, using rules similar to Self's existing multiple-inheritance resolution ones. The extra implicit receivers could be used to represent versions, preferences, or human agents on whose behalf an operation was being performed. Ungar and Smith posited that subjectivity was to object-oriented programming as object-oriented programming was to procedural programming. In procedural programming, the same function call always runs the same code; in object-oriented programming there is one object's worth of context (the receiver) and this object determines which code will run in response to a given message. In the US style of subjectivity, there are many objects' worth of context that determine what happens. Later on, others would coin the term "subject-oriented programming" to describe systems that were somewhat similar [HKKH96].

Slate combined prototypes with multiple dispatch [RS, SA05]. It strove to support a more dynamic object system than Cecil, and thus could support subjectivity as in US without compromising Self's dynamism. Slate put different ideas together in search of expressive power.

In contrast, Self inspired Antero Taivalsaari to simplify things even further. His **Kevo** language eschewed Self-style inheritance. Instead of sharing common state and behavior via special parent links, each Kevo object (at the linguistic level) contained all the state and behavior it could possibly exhibit [Tai93a, Tai92, Tai93]. Kevo's simplification of the runtime semantics of inheritance (i.e., no inheritance!) cast the dual problem of programming-time inheritance into sharp relief: Suppose a programmer needs to add a "union" method to every Set object. In Self, one can add it to a common parent. In Kevo, there were special operations to affect every object cloned from the same prototype. But these seemed to be too sensitive to the past; the operations relied on the cloning history, rather than whether an object was supposed to be a Set. For us, the Kevo language clarified the difference between the essential behavior needed to compute with an object and the cognitive structures needed to program (i.e., reflect upon) an object.

Getting back to something closer to Self, Jecel Assumpcao Jr.'s **Self/R** (a.k.a. **Merlin**) combined a Self-style language with a facility for low-level reflection in an effort to push the high-level language down into the operating system [Assu].

Moostrap was another language that adopted a Self-style object model as part of research into minimal languages based on reflection. Its name stood for Mini Object-Oriented System Towards Reflective Architectures for Programming [MC93, Mule95].

Lisac combined operating system research with programming language research [SC02]. The authors designed a language resembling Self with prototypes and dynamic inheritance and added some ideas from Eiffel. This language used static compilation and implemented an operating system; that is, it ran on bare metal. Dynamic inheritance was used in the video drivers and the file system.

SelfSync exploited the malleability of Self's object model to provide an interactive, bidirectional connection between an graphical diagram editor and a world of live, running, objects [PMH05]. It can be thought of as a visual programming language perched atop the Self system:

SelfSync is a Round-Trip Engineering (RTE) environment built on top of the object-oriented prototype-based language Self that integrates a graphical drawing editor for EER¹² diagrams. SelfSync realizes co-evolution between 'entities' in an EER diagram and Self 'implementation objects.' This is achieved by adding an extra EER 'view' to the default view on implementation objects in the model-view-controller [sic] architecture of Self's user interface. Both views are connected and synchronized onto the level of attributes and operations. [D'Hont]

Moving even further from the language design center of Self's creators, Self—though having no static type system whatsoever—inspired work on type systems for object-oriented languages. According to Stanford University professor John Mitchell: "The paper [FHM94] develops a calculus of objects and a type system for it. The paper uses a delegation-based approach and refers to your work on Self. This paper first appeared at the 1993 IEEE Symposium on Logic in Computer Science, and came before many other papers on type systems and object calculi. Abadi and Cardelli and many others also got involved in the topic at various times."

6.1.2. Distributed Programming Research

When Self was designed in 1986, computers were far less interconnected than they are today, and consequently the challenges of getting separate computers to work together have become far more important than they were in the '80s. When researchers tried to combine class-based objects and distributed programming, they discovered a problem: if two Point objects are to reside on separate computers, on which one should the Point class reside? On the one hand, since an object relies on its class to supply its behavior and interpretation, an object separated from its class is going to run very slowly. On the other hand, if the class data are replicated, then great effort must be expended to reconcile the conceptual chasm between a single, malleable class, and the reality of widespread replication of the class's contents. Along with the classless distributed system Emerald [BHJ86], Self's classless object model helped inspire researchers to consider such a model for a distributed system.

The closest such model to Self is probably **dSelf** [TK02], which adopted Self's syntax and object model, but let clones to reside on different machines and allowed an object to delegate to (i.e., be a child of) a parent object on a different machine.

12. EER: extended entity-relationship.

AmbientTalk was a classless distributed programming language designed for ad-hoc networks [DB04, Deid] that does not appear to have been directly influenced by Self.

Obliq, based on a prototype object model and dynamic typing, exploited the key concept of lexical scoping to provide secure, distributed, mobile computation [Card95].

6.1.3. Prototype Object Models for User Interface Languages and Toolkits

Self's legacy from ARK included the desire to bridge the gap between programming-level and graphical-level objects. In particular, prototypes seemed to be more concrete and easier to picture than classes. Perhaps that is why there have been some notable efforts to use a prototype-based object model for GUI languages and toolkits, including **Amulet**, which placed a prototype-based object model atop C++ to make it easier to build graphical interfaces [MMM97]:

Amulet includes many features specifically designed to make the creation of highly interactive, graphical, direct manipulation user interfaces significantly easier, including a prototype-instance object model, constraints, high-level input handling including automatic undo, built-in support for animation and gesture-recognition, and a full set of widgets. [Myer97]

Amulet was a follow-on to **Garnet**, which also used a prototype-instance object system [MGD90]. Although the authors do not tell us, this system may have been influenced by Self in that there is no distinction between instances and classes, data and methods are stored in “slots,” and slots that are not overridden by a particular instance inherit their values.

6.1.4. Other Impacts of the Self Language

In the late 1980s, a team at Apple Computer created one of the first commercial PDAs, the Apple Newton. Although the first products were marred by unreliable handwriting recognition—yes, the Newton aimed to recognize words the way one *naturally* wrote them—soon the Newton became an amazing device. It featured a simple intuitive interface with functionality that could be easily extended. What made it easy to build new Newton applications was its programming language, **NewtonScript**, a pure, object-oriented, dynamically typed language based on prototypes [Smi95] whose designer, Walter Smith, has cited Self as “one of the primary influences” [Smi94]. Like Self, NewtonScript created objects by cloning and had prioritized, object-based multiple inheritance. Unlike Self, slots were added to an object when assigned to, and each object had exactly two parents. Although the Newton was supplanted by the much smaller and lighter (but less flexible) Palm Pilot, it seems likely that the Newton was a key inspiration, so that Self was at least an indirect inspiration in the rise of PDAs.

Scripting Languages. Back when we designed Self, computers seemed to offer limitless power to those who could program them; we wanted to make this power available to the largest number of people, and thus we strove to lower programming’s cognitive barrier. Since then, computers have become ever faster and more widely used, trends that have created a niche for scripting languages. These notations were designed to be easy to learn and easy to use to customize systems such as web pages and browsers, but were not intended for large tasks in which performance was critical. In retrospect, it is not too surprising that many scripting languages were devised with object models like Self’s. The most popular of these by far seems to be **JavaScript** [FS02], which from the start was built into a popular Web

browser and has since become a standard for adding behavior to a Web page. JavaScript was based on a prototype model with object-based inheritance. Unlike Self, slots were added to an object upon assignment, reflective operations were not separated, and many more facilities were built into the language.

In addition to JavaScript, other prototype-based scripting languages have sprung up:

- **OScheme** is a small embeddable Scheme-like interpreter “that provides a prototype-based object model à la Self” [Bair].
- **Io** is a small, prototype-based programming language [Dek06]. More like Actors [Lieb86] than Self, its clones start out empty and gain slots upon assignment.
- **Glyptic Script** was a small, portable, and practical development environment and language that used both classes and prototypes [SL93, SLST94]. An object could be created by either instantiating a class or cloning an instance.
- After GlypticScript, Lentczner developed **Wheat**, a prototype-based programming system for creating of internet programs [Len05]: “Wheat strives to make programming dynamic web sites easy. It makes writing programs that span machines on the internet easy. It makes collaborative programming easy.” Wheat uses a tree object system instead of a heap, and each object has a URL. Its programming environment is a collaborative web site. Wheat’s design imaginatively melds object-oriented programming with distributed web-based objects.

Refactoring. Self’s simplicity can be a boon to automatic program manipulation tools. This simplicity may have encouraged Ivan Moore, a University of Manchester student working for Trevor Hopkins, to create Guru, a system that may well have been the first to automatically reorganize inheritance hierarchies to refactor programs while preserving their behavior [Moor, Moo95, Moo96, Moo96a, MC96]. Subsequent refactoring tools included the Refactoring Browser [RBJ97] for Smalltalk. Although a refactoring tool for Self would be even easier than for Smalltalk or Java, by the time refactoring tools became popular, the Self project was over.

6.1.5. Summary: Impact of Self Language

Self is still used by the authors, and Ungar has based his recent research on metacircular virtual machines on it. In addition, it is also in use by curious students from around the world and by a few other dedicated souls. Jecel Assumpcao in Brazil maintains an e-mail discussion list and a web site (there is one at Sun as well) from which the latest release can be downloaded. Volunteers have ported it to various platforms, and several language variants of Self have been designed. Still, the language cannot be said to be in widespread use; as of 2006 we estimate perhaps a dozen users on this planet.

Several pragmatic issues interfered with Self’s adoption in the early 1990s: the system was perceived as being too memory-hungry for its time, and too few people could afford the memory. Perhaps the worst problem was the challenge of delivering a small application instead of a large snapshot. The Self group was working on this when the project was cancelled in 1994: Ole Agesen’s work on type inferencing [AU94] showed promise in this area. Wolczko produced a standalone diff viewer that was half the speed of C, started in 1 second, and was correct (as opposed to the C version, which, according to Wolczko, was not). Finally, Self did not run on the most popular personal operating system of the time, Windows, and the complexity of the

virtual machine made a port seem like a daunting task for an outsider.

Self demonstrated that an object-oriented programming language need not rely on classes, that large programs could be built in such a fashion, and that it was possible to achieve high performance. These results helped free researchers to consider prototype-based linguistic structures [Blas94, DMC92, SLST94]. Of course, the languages in this section vary in their treatment of semantic issues like privacy, copying, and the role of inheritance. Yet all these languages have a model in which an object is in an important sense self-contained.

6.2. Implementation Techniques

The optimization techniques introduced by the Self virtual machine have served as a starting point for just about every desktop- and server-based object-oriented virtual machine today; for a nice survey, see [AFGH04]. The authors note that “the industry has invested heavily in adaptive optimization technology” and state that the Self implementation’s “technical highlights include polymorphic inline caches, on-stack-replacement, dynamic deoptimization, selective compilation with multiple compilers, type prediction and splitting, and profile-directed inlining integrated with adaptive recompilation.” Many subsequent virtual machines rely on these techniques. The survey’s authors also mention Self’s invocation count mechanism for triggering recompilation, and mention that the HotSpot Server VM, the initial IBM mixed-mode interpreter system, and the Intel Microprocessor Research Labs VM all used similar techniques. They point out that Self’s technique of deferring compilation of uncommon code has been adopted by the HotSpot server VM and the Jikes RVM, and that Self’s dynamic deoptimization technique that automatically reverts to deoptimized code for debugging “has been adopted by today’s leading production Java virtual machines.”

In a slightly more exuberant tone, Doederlein comments about the effect of (among other ideas) Self-style optimizations on Java performance: “The advents of Java2 (JDK1.2.0+), Sun HotSpot and IBM JDK, raised Java to previously undreamed-of performance, and has caught many hackers by surprise...Profile-based and Speculative JITs like HotSpot and IBM JDK are often seen as the Holy Grail of Java performance. [Hölzle’s *dissertation on the Self VM*] is the root of dynamic optimization” [Doe03]. (Italicized text added by present authors.)

The most direct influence of Self’s VM technology was on Sun’s HotSpot JVM, which is Sun’s Java desktop and server virtual machine and is used by other computer manufacturers including Apple Computer and Hewlett-Packard. It is an ironic story: In the fall of 1994, when the Self project was cancelled, two of Self’s people, Urs Hözle and Lars Bak, left Sun to join a startup, Animorphic Systems. (Hözle took a faculty position at UCSB and consulted at the startup; Bak was there full time.) The startup built Strongtalk, an impressive Smalltalk system that eventually included a virtual machine based on the Self virtual machine code base (with many improvements) and featuring an optional type system already designed by Animorphic’s Gilad Bracha and David Griswold [BG93]. Meanwhile, another Self alumnus, Ole Agesen at Sun Labs East, rewrote portions of Sun’s original JVM to support exact garbage collection.¹³ On the West Coast this project was nurtured by Mario Wolczko, another Self alumnus, who had written the clever feedback-mediated code to manage the Self garbage collector (see section 5.1). For a while, the Exact VM, as it was called, was Sun’s official JVM for Solaris. As Java became popular, Animorphic also retargeted its Smalltalk virtual machine to run

Java. Around this time, Bak and Hözle’s startup was acquired by Sun for its Java implementation and their Strongtalk system was left to languish. After the acquisition, Ungar (who had stayed at Sun all this time) loaned himself to the newly acquired group where he contributed the portability framework and the SPARC interpreter for Java. This virtual machine became HotSpot; HotSpot improved on Self by using an interpreter instead of a simple compiler for initial code execution, but retained the key techniques of adaptive optimization and dynamic deoptimization. HotSpot eventually became Sun’s primary virtual machine, supplanting the Exact VM. So the Self virtual machine essentially left the company, mutated somewhat, got reacquired, and now runs Java.

A bit later, there was talk within Sun of pushing on the debugging framework for Java. Smith, Wolczko, and others had the thought that surely the underlying code that allowed runtime method redefinition in the face of all those optimizations was laying there dormant in Sun’s HotSpot VM. (Recall that the HotSpot VM was originally created by modifying the Self VM.) Wolczko put Mikhail Dmitriev, then a Sun Laboratories intern (and later an employee) to work implementing method redefinition. With a working prototype in hand, Smith and Wolczko convinced the Sun Lab’s management to start the HotSwap project to allow fix-and-continue debugging changes. This facility is now part of the standard Sun Java VM, where it is used extensively for interactive profiling. According to Wolczko, this feature remains one key advantage of the NetBeans environment over its competition in 2006.

Other Java virtual machines have been inspired by the adaptive optimization and on-stack replacement in Self, including IBM’s Jalapeno (a.k.a. Jikes RVM) [BCFG99], [FQ03]. The JOEQ JVM has also been inspired by some techniques from Self, including what we called “deferred compilation of uncommon cases” [Wha01]. Adaptive optimization has even been combined with off-line profile information for Java [Krin03]. Although we have not been able to find any published literature confirming this, many believe that implementations of the .NET Common Language Runtime exploit some of these techniques.

Dynamic optimization and deoptimization also found applications removed from language implementation: Dynamo exploited adaptive optimization to automatically improve the performance of a native instruction stream [BDB00], and Transmeta used dynamic code translation and optimization to host x86 ISA programs on a lower-power microprocessor with a different architecture. Their code-morphing software may have been partially inspired by HotSpot [DGBJ03]. In addition to the Transmeta system, Apple computer’s Rosetta technology uses similar techniques to run PowerPC programs on Intel x86-based Macintosh computers [RRS99]. Moreover, as Ungar types these very letters, he is running a PowerPC word processor, FrameMaker, on an Intel-based MacBook Pro by using SheepShaver, a PowerPC emulator that exploits dynamic optimization [Beau06].

6.3. Cartoon Animation in the User Interface

Eleven years after their key paper on cartoon animation for user interfaces [CU93], Chang and Ungar won the second annual

13. Sun’s original “classic” JVM was not especially efficient, and relied on a garbage collection scheme that could not collect all garbage; it could be fooled into retaining vast amounts of space that were actually free. Such a scheme is called “conservative garbage collection” and was developed as a compromise for C-like systems that lack full runtime type information. This compromise was never essential for Java.

“Most Influential Paper” award for this work from the 2004 ACM Symposium on User Interface Software and Technology. Some of the influenced work includes the following:

- When researchers started working on immersive, 3D user interfaces, they built rapid prototyping environments. One such was Alice [CPGB94], whose creators found that the “same kind of ‘slow in/slow out’ animation techniques demonstrated in the Self programming system... (were) extremely useful in providing smooth state transitions of all kinds (position, color, opacity).”
- InterViews was a user interface toolkit for X-11 in C++. As part of the Prosodic Interfaces project, Thomas and Calder [TC95] took the notion of cartoon animation of graphical objects further, imbuing InterViews objects with elasticity and inertia. They stressed that such techniques meshed naturally with the goal of direct manipulation. In a subsequent paper [TC01], they went further and actually measured the effects of their animation techniques, showing them to be “effective and enjoyable for users.” Thomas and Demczuk used some of the same techniques to improve indirect manipulation, showing that animation could help users do alignment operations but that color and other effects were even better. Thomas has even applied cartoon animation techniques to a 3D collaborative virtual environment.
- Amulet [MMM97] incorporated an animation constraint solver that automatically animated the effects of changes to variables that denoted such things as positions and visibilities.
- Microsoft has studied the benefits of motion blur on the legibility of fast-moving cursors [BCR03]. They settled on temporal over-sampling. (We had seen this used in cartoons as well; Smith had christened it “stutter motion blur” as opposed to “streak motion blur.”)

Nowadays, although many aspects of cartoon animation can be found on commercial desktops—just click on the yellow button on your Macintosh OS X window to see squash and stretch—other aspects such as anticipation and followthrough remain to be exploited. OS X, though, does seem to have embraced the idea of smooth transitions, and some Microsoft systems also incorporate menus and text that fades in and out.

6.4. User Interface Frameworks

The principles of the Morphic UI have also been carried on into other interface frameworks, including one for Ruby [Ling04]. After Self ended, Maloney carried the Morphic GUI system into the Squeak version of Smalltalk [Mal01]. He followed the layout-as-morph approach with the AlignmentMorph class and its dozens of subclasses. Squeak’s current (2006) version of Morphic has diverged from Smith’s original architecture in that each morph includes a particular layout policy that is not a morph. However, because the policy is associated with a visible object rather than an often invisible AlignmentMorph, the newer design might be considered closer to Morphic principles. The AlignmentMorph class and its subclasses are used in the latest version of Squeak, and informal discussions with Squeak users give us a sense that the proper way to treat the GUI visual structuring problem is still debated.

7. Looking Back

Now that the world has seen Self and we have received the benefit of hindsight, we can comment on lessons learned and interesting issues.

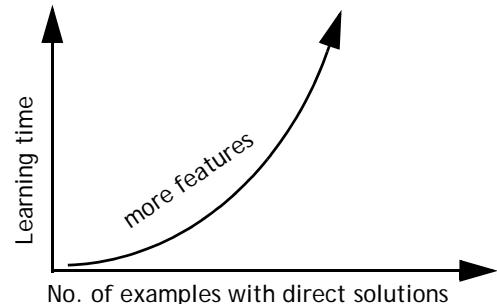


Figure 26. As more features are embedded in the language, the programmer gets to do more things immediately. But complexity grows with each feature: how the fundamental language elements interact with one another must be defined, so complexity growth can be combinatorial. Such complexity makes the basic language harder to learn, and can make it harder to use by forcing the programmer to make a choice among implementation options that may have to be revisited later.

7.1. Language

Minimalism. Ungar confesses, with some feelings of guilt, that the pure vision of Self suffered at his own hands, as he yielded to temptation and tried adding a few extra features here and there. But how could the temptation to feature creep seduce members of the Self team, who so vocally extol the principles of uniformity and simplicity? Looking back, we think it arose from the siren song of the well-stated example. Ungar had to learn the hard way that smaller was better and that examples could be deceptive. Early in the evolution of Self he made three such mistakes: prioritized multiple inheritance, the sender-path tie-breaker rule, and method-holder-based privacy semantics.¹⁴ Each was motivated by a compelling example [UCH91]. We prioritized multiple parent slots to support a mix-in style of programming. The sender-path tiebreaker rule allowed two disjoint objects to be used as parents, for example a rectangle parent and a tree-node parent for a VLSI cell object. The method-holder-based privacy semantics allowed objects with the same parents to be part of the same encapsulation domain, thereby supporting binary operations in a way that Smalltalk could not [UCH91].

But each feature also caused no end of confusion. The prioritization of multiple parents implied that Self’s “resend” (call-next-method) lookup had to be prepared to back up along parent links to follow lower-priority paths. The resultant semantics took five pages to write down, but we persevered. As mentioned in section 4.2, after a year’s experience with the features, we found that each of the members of the Self group had wasted no small amount of time chasing “compiler bugs” that were merely their unforeseen consequences. It became clear that the language had strayed from its original path. Ironically, Ungar, who had once coined the term “architect’s trap” for something similar in computer architecture, fell right into what might be called “the language designer’s trap.” He is waiting for the next one. At least in computer architecture and language design, when features, rules, or elaborations are motivated by particular examples, it is a good bet that their addition will be a mistake.

Prototypes and Classes. Prototypes are often presented as an alternative to class-based language designs, so the subject of

14. In all fairness, recall that Smith was across the Atlantic at the time and so, on the one hand, had nothing to do with these mistakes. On the other hand, Ungar chides him that if he had not wandered off, maybe such mistakes could have been avoided.

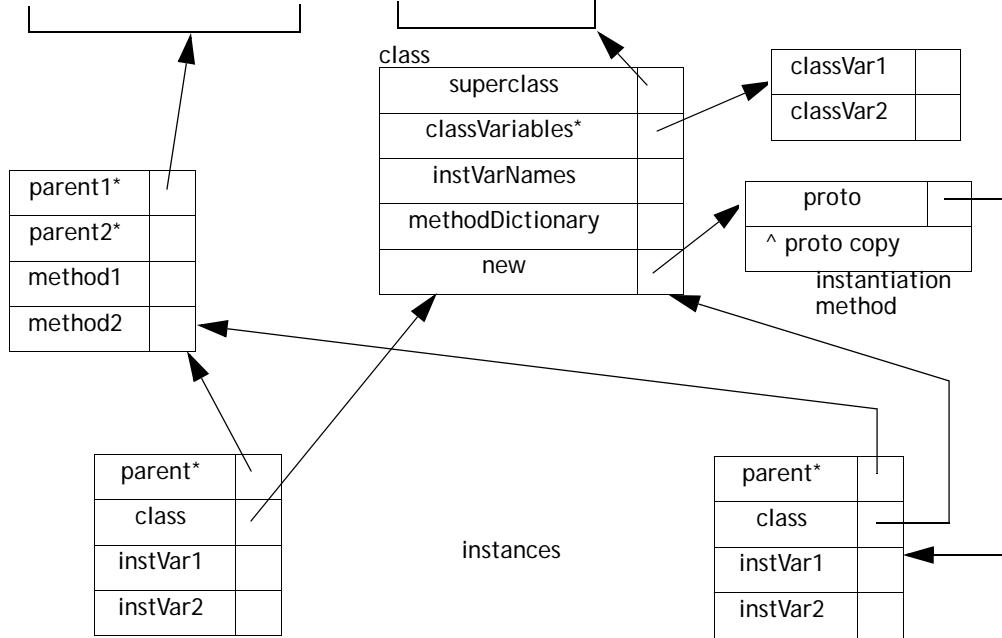


Figure 27. This figure suggests how Self objects might be composed to form Smalltalk-like class structures (demonstrated more completely by Wolczko [Wol96]). He shows that, with some caveats, Smalltalk code can be read into a Self system, parsed into Self objects, then executed with significant performance benefits, thanks to Self’s dynamically optimizing virtual machine.

prototypes vs. classes can serve as point of (usually good natured) debate.

In a class-based system, any change (such as a new instance variable) to a class affects new instances of a subclass. In Self, a change to a prototype (such as a new slot) affects nothing other than the prototype itself (and its subsequent direct copies).¹⁵ So we implemented a “copy-down” mechanism in the environment to share implementation information. It allowed the programmer to add and remove slots to and from an entire hierarchy of prototypes in a single operation. Functionality provided at the language level in class-based systems rose to the programming environment level in Self. In general, the simple object model in Self meant that some functionality omitted from the language went into the environment. Because the environment is built out of Self objects, the copy-down policy can be changed by the programmer. But such flexibility incurred a cost: there were two interfaces for adding slots to objects, the simple language level and the copying-down Self-object level. This loss of uniformity could be confusing when writing a program that needs to add slots to objects. Although we managed fine in Self, if Ungar were to design a new language, he might be tempted to include inheritance of structure in the language, although it would still be based on prototypes. Smith remains unconvinced.

A brief examination of the emulation of classes in Self illuminates both the nature of a prototype-based object model and the tradeoff between implementing concepts in the language and in the environment. To make a Self shared parent look more like a class, one could create a “new” method in the shared parent. This method could make a copy of some internal reference to a prototype, and so would appear to be an instantiation device.

15. Self prototypes are not really special objects, but are distinguished only by the fact that, by convention, they are copied. Any copy of the prototype would serve as a prototype equally well. Some other prototype-based systems took a different approach.

Figure 27 suggests how to make a Smalltalk class out of Self objects. Mario Wolczko built a more complete implementation of this, and showed [WAU96, Wol96] that it worked quite well: he could read in Smalltalk source code and execute it as a Self program. There are certain restrictions on the Smalltalk source but, thanks to Self’s implementation technology, once the code adaptively optimizes, the Self version of Smalltalk code generally ran faster than the Smalltalk version! General meta-object issues in prototype-based languages were tackled by the Moostrap system [Mule95].

The world of Self objects and how they inherit from one another results in a roughly hierachal organization, with objects in the middle of the hierarchy tending to act as repositories of shared behavior. Such behavior repositories came to be called “traits” objects.¹⁶ The use of traits is perhaps only one of many ways of organizing the system, and may in fact have been a carryover from the Self group’s Smalltalk experience. Interestingly, it is likely that our old habits may not have done Self justice (as observed in [DMC92]). Some alternative organizational schemes might have avoided a problem with the traits: a traits object cannot respond to many of the messages it defines in its own slots! For example, the point traits object lacked `x` and `y` slots and so could not respond to `printString`, since its `printString` slot contained a method that in turn sent `x` and `y` messages. We probably would have done better to put more effort into exploring other organizations. When investigating a new language, one’s old habits can lead one astray.

Another problem plaguing many prototype-based systems is that of the corrupted prototype. Imagine copying the prototypical

16. Do not confuse these traits objects with the construction written about in the past few years [SDNB03]. These had nothing to do with and predated by many years the more recent use of the word “traits” in object-oriented language design.

list, asking it to print, then finding it not empty as expected, but already containing 18 objects left there by some previous program! Self's syntax makes the previous program's mistake somewhat easy: the difference between

```
list copy add: someObj ect.
```

and

```
list add: someObj ect.
```

is that the latter puts some object in the system's prototypical list.

Addressing this problem led to some spirited debate within the Self group. Should the programmer have the right to assume anything about the prototypical list? It is, after all, just another object. To the VM, yes, but to the programmer, it is quite a distinguished object. Our solution, though disturbing at some level, was to introduce a `copyRemoveAll` method for our collections. Use of this method guaranteed an empty list, yet it was clearly only a partial solution. What if some program, now long gone, accidentally uttered the expression:

```
Date currentYear: 1850
```

This would create trouble for any program that subsequently assumed the current year was properly initialized in copies of the Date object (unless it was running inside a time machine).

As we have said many times by now, when designing Self we sought to unify assignment and computation. This desire for access/assignment symmetry could be interpreted as arising from the sensory-motor level of experience. Lakoff and Johnson put it very well [LJ87], although we had not read their work at the time we designed Self: from the time we are children, experience and manipulation are inextricably intertwined; we best experience an object when we can touch it, pick it up, turn it over, push its buttons, even taste it. We believe that the notion of a container is a fundamental intuition that humans share and that by unifying assignment and computation in the same way as access and computation, Self allows abstraction over containerhood. Since all containers are inspected or filled by sending messages, any object can pretend to be a container while employing a different implementation.

Retrospective Thoughts on the Influence of Smalltalk. In writing this paper and looking over the principles of Smalltalk enumerated by Ingalls [Inga81], we realize that in most cases we tried to take them even further than Smalltalk did. Table 2 shows, for each of Ingalls' principles, the progression from Smalltalk through ARK to Self.

Table 2: Ingalls' Principles of Programming System Design

Principle	Smalltalk	ARK	Self
Personal Mastery	If a system is to serve the creative spirit, it must be entirely comprehensible to a single individual.	A primary goal of ARK was to make possible personal construction of alternate realities, increasing comprehension by tangibly manifesting objects in the UI.	Concepts such as classes were removed to get a simpler language.
Good Design	A system should be built with a minimum set of unchangeable parts; those parts should be as general as possible; and all parts of the system should be held in a uniform framework.	ARK contains the world of Smalltalk objects, any of which could appear as a simulated tangible object with mass and velocity (e.g., it was possible to grab the number 17 and throw it into orbit around a simulated planet).	Even the few operations that were hard-wired in Smalltalk, such as integer addition, identity comparison, and basic control structures such as "ifTrue:" are user-definable in Self.
Objects	A computer language should support the concept of "object" and provide a uniform means for referring to the objects in its universe.	In ARK, the uniform means for referring to the objects mentioned in this principle included object references used in Smalltalk, but ARK added something with its ability to represent any object inside an alternate reality. In other words, uniformity of object access was passed up into the UI as well.	A Self object is self-sufficient; no class is needed to specify an object's structure or behavior.
Storage Management	To be truly object-oriented, a computer system must provide automatic storage management.		Generational, nondisruptive garbage collection for young objects and feedback-mediated mark-sweep for old objects.

Table 2: Ingalls' Principles of Programming System Design (Continued)

Principle	Smalltalk	ARK	Self
Messages	Computing should be viewed as an intrinsic capability of objects that can be uniformly invoked by sending messages.	ARK, as Self would later, replaced the notion of variable access with message sends. Hence it might be seen as taking this principle even further.	Smalltalk includes both message passing and variable access/assignment in its expressions. Self expressions includes <i>only</i> message passing; “variables” are realized by pairs of accessor/assignment methods.
Uniform Metaphor	A language should be designed around a powerful metaphor that can be uniformly applied in all areas.	ARK took the metaphor of object and message inherited from the underlying Smalltalk level and pushed it up into the UI, in that every object could be manipulated as a tangible object on the screen with physical attributes such as mass and velocity, suitable for a simulated world.	Self includes no separate scoping rules, and reuses objects and inheritance instead of Smalltalk’s special-purpose system and method dictionaries.
Modularity	No component in a complex system should depend on the internal details of any other component.		Self followed Smalltalk in restricting base-level access to other objects to only message-passing. However, Smalltalk includes messages, inherited by every class, that allow one object to inspect the internals of another (e.g., “instanceVariableAt:”). Self improves on Smalltalk’s modularity by separating this facility into a separate reflection protocol, implemented by mirror objects. This facility can be disabled by turning off the one virtual machine primitive that creates mirror objects.
Classification	A language must provide a means for classifying similar objects, and for adding new classes of objects on equal footing with the system’s kernel classes.	ARK did not pay much attention to classification issues. New kinds of objects could be made by adding new state to some existing instance, but they were anonymous, so the user did not even have a name to go on. The Smalltalk categories used in the browser were also used in the Alternate Reality Kit’s “warehouse” icon, which strove to make an instance of any class selected from the warehouse’s pop-up hierarchical menu.	Self has no classes. We did not find them essential, opting to supply such structure at higher levels in the system.
Polymorphism	A program should specify only the behavior of objects, not their representation.		In Self, even the code “within” an object is isolated from the object’s representation.
Factoring	Each independent component in a system should appear in only one place.		Self’s prototype model, which did not build inheritance of structure into the language, simplifies the specification of multiple inheritance.

Table 2: Ingalls' Principles of Programming System Design (Continued)

Principle	Smalltalk	ARK	Self
Virtual Machine	A virtual machine specification establishes a framework for the application of technology.		
Reactive Principle	Every component accessible to the user should be able to present itself in a meaningful way for observation and manipulation.	One of the main goals of ARK was to make objects feel more real, more directly present. This can be seen as an attempt to (as in the original articulation of this principle) “show the object in a more meaningful way for observation and manipulation.”	The Morphic User Interface improved upon the Smalltalk-80 UI. In Smalltalk, scroll bars and menus could not be graphically selected, only used. In Self’s Morphic they can. (Of course, Self has the luxury of a more powerful platform.)

7.2. Implementation techniques

The efficacy of the Self VM in obtaining good performance for a dynamic, purely object-oriented language came at a high price in complexity and maintainability. One issue that has arisen since the original optimization work has been the difficulty of finding intermittent bugs in a system that adaptively optimizes and replaces stack frames behind the user’s back. Since 1995, the Self virtual machine has been maintained primarily by Ungar in his spare time, so the priorities have shifted from probing the limits of performance to reducing maintenance time. Consequently, when we run Self today, we disable on-stack replacement.

Looking back, it’s clear that the optimizations devised for Self were both the hardest part of the project, spanning many years and several researchers, and also—despite their complexity—it’s most widely adopted part of the project. This experience argues for stubborn persistence on the part of researchers and a large dose of patience on the part of research sponsors.

7.3. UI2 and Morphic

On the whole we were satisfied with much of UI2. While the principles of live editing and structural reification helped create the sense of working within a world of tangible yet malleable objects, we could imagine going further. Several things interfered with full realization of those goals.

Multiple views. The very existence of the outliner as a separate view of a morph object weakened the sense of directness we were after. After all, when one wanted to add a slot to an object, one had to work on a different display object, the object’s outliner. We never had the courage or time to go after some of the wild ideas that would have made possible the unification of any morph with its outliner. Ironically, Self’s first interface, UI1, probably did better in this respect because it limited itself to presenting only outliners.

Text and object. There is a fundamental clash between the use of text and the use of direct manipulation. A word inherently denotes something, an object does not necessarily denote anything. That is, when you see the word “cow,” an image comes to mind, an image totally different from the word “cow” itself. It is in fact difficult to avoid the image: that is the way that words are supposed to work. Words stand for things, but a physical object does not necessarily stand for anything. Textual notation and object manipulation are fundamentally from two different realities.

Text is used quite a bit in Self, and its denotational character weakens the sense of direct encounter with objects. For example, many tools in the user interface employed a “printString” to denote an object. The programmer working with one of these tools might encounter the text “list (3, 7, 9).” The programmer might know that this denoted an object which could be viewed “directly” with an outliner. But why bother? The textual string often says all one needs to know. The programmer moves on, satisfied perhaps, yet not particularly feeling as if they encountered the list itself. The mind-set in a denotational world is different from that in a direct object world, and use of text created a different kind of experience. Issues of use and mention in direct manipulation interfaces were discussed further [SUC92].

8. Conclusion

Shall machines serve humanity, or shall humanity serve machines? People create artifacts that then turn around and reshape their creators. These artifacts include thought systems that can have profound effects, such as quantum mechanics, calculus, and the scientific method. In our own field thought systems with somewhat less profound effects might include FORTRAN and Excel. Some thought systems are themselves meta-thought systems; that is, they are ways of thinking followed when building other thought systems. Since they guide the construction of other thought systems, their impact can be especially great, and one must be especially careful when designing such meta-thought systems.

We viewed Self as a meta-thought system that represented our best effort to create a system for computer programming. The story of its creation reveals our own ways of thinking and how other meta-thought systems shaped us [US87, SU95]. We kept the language simple, built a complicated virtual machine that would run programs efficiently even if they were well-factored, and built a user interface that harnessed people’s experience in dealing with the real world to off load conscious tasks to pre-cognitive mental facilities. We did all of this in the hope that the experience of building software with the Self system would help people to unleash their own creative powers.

However, we found ourselves trying to do this in a commercial environment. Free markets tend to favor giving customers what they want, and few customers could then (or even now) understand that they might want the sort of experience we were creating.

Years later, the Self project remains the authors’ proudest professional accomplishment. We feel that Self brought new ideas to language, implementation, programming environment, and

graphical interface design. The original paper [US87] has been cited over 500 times, and (as previously mentioned) received an award at the OOSPLA 2006 conference for among the three most influential conference papers published during OOPSLA's first 11 years (1985 through 1996). Self shows how related principles can be combined to create a pure, productive, and fun experience for programmers and users.

So, what happened? Why isn't your word processor written in Self? While we have discussed the struggle of ideas that gave birth to Self, we have not addressed the complex of forces that lead to adoption (or not) of new technology. The implementation techniques were readily adopted, whereas semantic notions such as using prototypes, and many of the user interface ideas behind Morphic, were not so widely adopted. We believe that, despite the pragmatic reasons mentioned in section 6.1.5, this discrepancy can better be explained by the relative invisibility of the virtual machine. If there are dynamic compilation techniques going on underneath their program, most users are unlikely to know or care. But the user interface framework and the language semantics demand that our users think a certain way, and we failed to convince the world that our way to think might be better. Did our fault lie in trying to enable a creative spirit that we mistakenly thought lay nascent within everybody? Or are there economic implications to the use of dynamic languages that make them unrealistic? Many of us in the programming language research community secretly wonder if language research has become irrelevant to most of the world's programmers, despite the obvious truth that in many ways, computers remain painful, opaque black boxes that at times seem intent on spreading new kinds of digital pestilence.

Almost two decades after the conception of Self, the imbalance of power between man and machine seems little better. We are still waiting for computers to begin to live up to their full promise of being a truly malleable and creative medium. We earnestly hope that Self may inspire those who still seek to simplify programming and to bring it into coherence with the way most people think about the real world.

9. Epilogue: Where Are They Now?

After the Self project, the people involved followed disparate paths. Smith, Ungar, and Wolczko stayed at Sun Laboratories. Randy Smith used Morphic's shared space aspects to start a project studying distance learning. He also worked on realtime collaboration support for Java, then researched user interfaces techniques for information visualization. Randy now works on trying to make it easier to understand and use sensor networks. He continues to use Self for an occasional quick prototype, especially when a live shared-space demo would be useful.

David Ungar has used Self in much of his research. With help from Michael Abd-El Malek, the complex Self virtual machine was ported to the Macintosh computer system. David also worked on Sun's HotSpot Java virtual machine, and until recently was researching Klein, a meta-circular VM architecture in Self for Self [USA05].

After a brief flirtation with binary translation, Mario Wolczko worked on Sun's ExactVM (a.k.a. Solaris Production Release of Java 1.2 JVM), then managed the group that developed the research prototype for Sun's KVM, a Java VM for small devices. Since then he has been working on architecture support for Java, automatic storage reclamation, and objects, as well as performance monitoring hardware for various SPARC microprocessors at Sun Microsystems Laboratories.

Elgin Lee went to ParcPlace Systems, and now does legal consulting.

Lars Bak left Sun to build a high performance virtual machine for Smalltalk at the startup Animorphic Systems. The technology was adapted to Java, and the Java HotSpot system was born. Sun acquired the startup and Bak ended up leading the HotSpot project until it successfully shipped in 1997. Next, Bak designed a lean and mean Java virtual machine for mobile phones, commercialized by Sun as CLDC HI. Bak left Sun again to pursue even smaller virtual machines. The startup OOVM was founded to create an always running Smalltalk platform for small embedded devices. The platform had powerful reflective features despite a memory footprint of 128KB. OOVM was acquired by Esmertec AG.

After the Self project, Ole Agesen implemented a Java-to-Self translator that, for a time, seemed to be the world's fastest Java system. Then he spearheaded a project at Sun incorporating exact garbage collection into Sun's original JVM; after that, he went to VMware, working on efficient software implementations of x86 CPUs. Many of the implementation techniques successful in dynamic languages can be reused for x86: it is really just a different kind of bytecode (x86) that is translated. More specifically, Ole has been on the team that designed and implemented the SMP version of VMware; more recently, he has worked on supporting 64-bit ISAs (x86-64).

Since graduating from Stanford, Craig Chambers has been a professor at the University of Washington, where he worked on language designs including Cecil, MultiJava, ArchJava, and EML, and on optimizing compiler techniques primarily targeting object-oriented languages. The language designs were inspired by Self's high level of simplicity and uniformity, while also incorporating features such as multiple dispatching and polymorphic, modular static type checking. The optimizing compiler research directly followed the Self optimizing dynamic compiler research, in some cases exploring alternative techniques such as link-time whole-program compilation as in the Vortex compiler, and in other cases applying (staged) dynamic compilation to languages such as C, as in the DyC project.

After Self, Bay-Wei Chang was at PARC for four years working on document editing, annotating, and reading interfaces for web, desktop, and mobile devices. For the past six years, Chang has been at the research group at Google working on bits of everything, including web characterization, mobile interfaces, e-mail interfaces, web search interfaces and tools, and advertising tools.

After Self, Urs Hözle was at UCSB from 1994-99 as Assistant/Associate Professor. During that time, he also worked part-time with Lars Bak, first at Animorphic and then at Sun's Java organization on what became Sun's HotSpot JVM. Since 1999 Hözle has been at Google in various roles (none involving dynamic compilation to date!), first as search engine mechanic and later as VP Engineering for search quality, hardware platforms, and as VP of Operations.

From the Self group, John Maloney went to work for Alan Kay for about six years, first at Apple's Advanced Technology Group and then at Walt Disney Imagineering R&D. (Alan Kay moved the entire group from Apple to Disney.) While there, John helped implement the Squeak Virtual Machine, notable because the VM itself was written (and debugged) in Smalltalk, then automatically translated into C code for faster execution. This technique resulted in an extremely portable, stable, and platform-independent virtual machine. Once they had the VM, John re-implemented Morphic in Smalltalk with very few design

changes from the Self version. The Morphic design has stood the test of time and has enabled a rich set of applications in Squeak, including the EToys programming system for children. In October 2002, John moved to the Lifelong Kindergarten Group at the MIT Media Lab, where he became the lead programmer for Scratch, a media-rich programming system for kids. Scratch is built on top of Squeak and Morphic. Is it currently in beta testing at sites around the world and will become publicly available in the summer of 2006.

10. Acknowledgments

The Self project resulted from the inspired creation of many individuals and the funding of many organizations, as mentioned in the Introduction. This lengthy paper about the Self project has resulted not only from the authors' efforts, but from the information and advice gleaned from many individuals. We especially wish to acknowledge the review committee chaired by Kathleen Fisher: each reviewer commented in great detail, most of them several times—our heartfelt thanks to each of you. Your diligence spoke of your desire to see a high-quality paper on the history of Self, and that desire in turn inspired us through the many arduous hours. Our special thanks to committee member Andrew Black, who was particularly thorough and thoughtful in his copious comments and suggestions.

11. Appendix: Self Release Announcements

11.1. Self 2.0

What: Self 2.0 Release

From: hoelzle@Xenon.Stanford.EDU (Urs Hoelzle)

Date: 10 Aug 92 21:08:25 GMT

Announcing Self Release 2.0

The Self Group at Sun Microsystems Laboratories, Inc., and Stanford University is pleased to announce Release 2.0 of the experimental object-oriented exploratory programming language Self.

Release 2.0 introduces full source-level debugging of optimized code, adaptive optimization to shorten compile pauses, lightweight threads within Self, support for dynamically linking foreign functions, changing programs within Self, and the ability to run the experimental Self graphical browser under OpenWindows.

Designed for expressive power and malleability, Self combines a pure, prototype-based object model with uniform access to state and behavior. Unlike other languages, Self allows objects to inherit state and to change their patterns of inheritance dynamically. Self's customizing compiler can generate very efficient code compared to other dynamically-typed object-oriented languages.

Self Release 2.0 runs on Sun-3's and Sun-4's, but no longer has an optimizing compiler for the Sun-3 (and therefore runs slower on the Sun-3 than previous releases).

This release is available free of charge and can be obtained via anonymous ftp from self.stanford.edu. Unlike previous releases, Release 2.0 includes all source code and is legally unencumbered (see the LICENSE file for legal information.) Also available for ftp are a number of papers published about Self.

Finally, there is a mail group for those interested in random ramblings about Self, self-interest@self.stanford.edu. Send mail to self-request@self.stanford.edu to be added to it (please do not send such requests to the mailing list itself!).

The Self Group at Sun Microsystems Laboratories, Inc. and Stanford University

11.2. Self 3.0

From: hoelzle@Xenon.Stanford.EDU (Urs Hoelzle)

Subject: Announcing Self 3.0

Date: 28 Dec 93 22:19:34 GMT

ANNOUNCING Self 3.0

The Self Group at Sun Microsystems Laboratories, Inc., and Stanford University is pleased to announce Release 3.0 of the experimental object-oriented programming language Self. This release provides simple installation, and starts up with an interactive, animated tutorial.

Designed for expressive power and malleability, Self combines a pure, prototype-based object model with uniform access to state and behavior. Unlike other languages, Self allows objects to inherit state and to change their patterns of inheritance dynamically. Self's customizing compiler can generate very efficient code compared to other dynamically-typed object-oriented languages.

The latest release is more mature than the earlier releases: more Self code has been written, debugging is easier, multiprocessing is more robust, and more has been added to the experimental graphical user interface which can now be used to develop code. There is now a mechanism (still under development) for saving objects in modules, and a source-level profiler.

The Self system is the result of an ongoing research project and therefore is an experimental system. We believe, however, that the system is stable enough to be used by a larger community, giving people outside of the project a chance to explore Self.

2 This Release

This release is available free of charge and can be obtained via anonymous ftp from Self.stanford.edu. Also available for ftp are a number of published papers about Self. There is a mail group for those interested in random ramblings about Self, Self-interest@Self.stanford.edu. Send mail to self-request@self.stanford.edu to be added to it (please do not send such requests to the mailing list itself!).

2.1 Implementation Status

Self currently runs on SPARC-based Sun workstations running SunOS 4.1.x or Solaris 2.3. The Sun-3 implementation is no longer provided.

2.2 Major Changes

Below is a list of changes and enhancements that have been made since the last release (2.0.1). Only the major changes are included.

- The graphical browser has been extended to include editing capabilities. All programming tasks may now be performed through the graphical user interface (the "ui"). Type-ins allow for expression evaluation, menus support slot editing, and methods can be entered and edited. If you are familiar with a previous version of the Self system, Section 14.1 of the manual entitled "How to Use Self 3.0" contains a quick introduction to the graphical user interface. The impatient might want to read that first.
- A mechanism - the transporter - has been added to allow arbitrary object graphs to be saved into files as Self source. The system has been completely modularized to use the transporter; every item of source now resides in a trans-

Let's add some new behavior to this tank of an ideal gas.
Press the 'start' button to see the atoms bounce around.

start

Get the right button menu on the gas tank, and select 'Outliner for Morph...'

Hit the little black triangle to expand the outliner. The outliner is a representation of the gas tank as a Self object. Some of the slots can be found inside the category tree (Basic Morph State, Frame Morph State, etc.).

```

An object(type: gasTank)
Modules: idealGas, -, morphSaving
parent# traits gasTank =
frictionPercentage 100 ...
heatUp
atoms do: [I:=al
    a velocity: a velocity * 1.5
].
self
▶ Basic Morph State
▶ Frame Morph State
▶ firing out

```

porter-generated module. Transport-generated files have the suffix .sm to distinguish them from “handwritten” files (.Self), though this may change as we move away from handwritten source. The transporter is usable but rough, we are still working on it.

- Every slot or object may now have an annotation describing the purpose of the slot. In the current system, annotations are strings used to categorize slots. We no longer categorize slots using explicit category parent objects. Extra syntax is provided to annotate objects and slots.
- A new profiler has been added, which can properly account for the time spent in different processes and the run-time system, and which presents a source-level profile including type information (i.e., methods inherited by different objects are not amalgamated in the profile, nor are calls to the same method from different sites). It also presents a consistent source-level view, abstracting from the various compiler optimizations (such as inlining) which may confuse the programmer.
- Privacy is not enforced, although the privacy syntax is still accepted. The previous scheme was at once too restrictive (in that there was no notion of “friend” objects) and too lax (too many object had access to a private slot). We hope to include a better scheme in the next release.
- The “new” compiler has been supplanted by the SIC (“simple inlining compiler”), and the standard configuration of the system is to compile first with a fast non-optimizing compiler and to recompile later with the SIC. Pauses due to compilation or recompilation are much smaller, and applications usually run faster.
- Characters are now single-byte strings. There is no separate character traits.
- Prioritized inheritance has been removed; the programmer must now manually resolve conflicts. We found the priority mechanism of limited use, and had the potential for obscure errors.

2.4 Bug Reports

Bug reports can be sent to self-bugs@self.stanford.edu. Please include an exact description of the problem and a short Self program reproducing the bug.

2.5 Documentation

This release comes with two manuals:
How to Use Self 3.0 (SelfUserMan.ps)
The Self Programmer’s Reference Manual (progRef.ps)

Happy Holidays!

-- The Self Group

11.3. Self 4.0

Below is a redacted form of the Self 4.0 release announcement made on July 10, 1995. The text we do include has not been edited.

The [Self Group](#) at [Sun Microsystems Laboratories, Inc.](#), and [Stanford University](#) has made available Release 4.0 of the experimental object-oriented programming language [Self](#).

This release of Self 4.0 provides simple installation, and starts up with an interactive, animated tutorial (a small piece of which is shown below).

Self 4.0 is, in some sense, the culmination of the Self project, which no longer officially exists at Sun. It allows novices to start by running applications, smoothly progress to building user interfaces by directly manipulating buttons, frames and the like, progress further to altering scripts, and finally to ascend to the heights of complete collaborative application development, all without ever stumbling over high cognitive hurdles.

Its user interface framework features automatic continuous layout, support for ubiquitous animation, direct-manipulation-based construction, the ability to dissect any widget you can see, and large, shared, two-dimensional spaces.

Its programming environment is based on an outliner metaphor, and features rapid turnaround of programming changes. It includes a plethora of tools for searching the system. Its debugger supports in-place editing. A structure editor supports some static type checking and helps visualize complex expressions. Finally, the programming environment features the new transporter, which eases the task of saving programs as source files.

Self 4.0 includes two applications: an experimental web browser, and an experimental Smalltalk system.

Major Changes in Self 4.0.

Below is a list of changes and enhancements that have been made since the last release (4.0). Only the major changes are included.

- This release contains an entirely new user interface and programming environment which enables the programmer to create and modify objects entirely within the environment, then save the objects into files. You no longer have to edit source files using an external editor. The environment includes a graphical debugger, and tools for navigation through the system.

- Any Self window can be shared with other users on the net: users each have their own cursor, and can act independently to grab and manipulate objects simultaneously. A Self window is actually a framed view onto a vast two-dimensional plane: users can move their frames across this surface, bringing them together to work on the same set of objects, or moving apart to work independently.
- A new version of the transporter, a facility for saving objects structure into files, has been used to modularize the system. The programming environment presents an interface to the module system which allows for straightforward categorization of objects and slots into modules, and the mostly-automatic saving of modules into files. Handwritten source files have almost completely disappeared.
- The environment has been constructed using a new, flexible and extensible user interface construction kit, based on “morphs.” Morphs are general-purpose user interface components. An extensive collection of ready-built morphs is provided in the system, together with facilities to inspect, modify, and save them to files. We believe the morph-based substrate provides an unprecedented degree of directness and flexibility in user interface construction.
- An experimental Web browser has been written in Self and is included in the release. This browser supports collaborative net-surfing, and the buttons and pictures from Web pages can easily be removed and embedded into applications.
- A Smalltalk system is included in Self 4.0. This system is based on the GNU system classes, a translator that reads Smalltalk files and translates them to Self, and a Smalltalk user interface. The geometric mean of four medium-sized benchmarks we have tried suggests that this system runs Smalltalk programs 1.7 times faster than commercially available Smalltalk on a SPARCstation.
- Significant engineering has been done on the Virtual Machine to reduce the memory footprint and enhance memory management. For example, a 4.0 system containing a comparable collection of objects to that in the 3.0 release requires 40% less heap space. A SELF-level interface to the memory system is now available that enables SELF code to be notified when heap space is running low, and to expand the heap.
- The privacy syntax has been removed; in the previous release it was accepted but privacy was not enforced. The concept of privacy still exists, and is visible in the user interface, but is supported entirely through the annotation system.

SELF currently runs on SPARC-based Sun workstations using Solaris 2.3 or later, or SunOS 4.1.x. The compiler is an improved version of the one used in 3.0.

System requirements. To run SELF you will need a SPARC-based Sun computer or clone running SunOS 4.1.X or Solaris 2.3 or 2.4.

To use the programming environment you will need to run X Windows version 11 or OpenWindows on an 8-bit or deeper color display. The X server need not reside on the same host as SELF.

The SELF system as distributed, with on-line tutorial, Web browser and Smalltalk emulator, requires a machine with 48Mb of RAM or more to run well.

The user interface makes substantial demands of the X server. A graphics accelerator (such as a GX card) improves the responsiveness of the user interface significantly, and therefore we recommend that you use one if possible.

We hope that you enjoy using Self as much as we do.

-- The Self Group July 10, 1995

11.4. Self 4.3 (The latest release as of 2006)



The Power of Simplicity
Release 4.3

Adam Spitz, Alex Ausch, and David Ungar
Sun Microsystems Laboratories

June 30, 2006

Late-breaking news. Self now runs under Intel-based Macintoshes (as well as PowerPC-based and SPARCTM-based systems), though it does not yet run on Windows or Linux. Additionally, the original Self user interface (UI1) has been resurrected, although its cartoon-animation techniques have not yet been incorporated into the default Self user interface (UI2). See the included release notes for a full list of changes.

Downloading. If you want to run Self 4.3, download and unpack one of the following:

- Self 4.3 for Mac OS X in compressed disk image format, or
- Self 4.3 for SPARCTM workstations from Sun Microsystems running the Solaris™ operating system in tar/gzip format

See the release notes for directions on how to run Self. (We're hoping that the procedure is fairly self-explanatory, though. If it's not, please contact us!)

If you also want to work on the Self virtual machine (most users will not want to do this), you will need to download one of the above packages, and you will also need one of the following:

- Virtual machine and Self sources in compressed disk image format, or
- Virtual machine and Self sources in tar/gzip format

What Self is. Self is a prototype-based dynamic object-oriented programming language, environment, and virtual machine centered around the principles of simplicity, uniformity, concreteness, and liveness. It was developed by the Self Group at Sun Microsystems Laboratories, Inc. and Stanford University.

Although Self is no longer an official project at Sun Microsystems Laboratories, we have seen many of Self's innovations adopted. The Morphic GUI framework has been incorporated into Squeak, and the virtual machine ideas provided the initial inspiration for the Java™ HotSpot™ performance engine. However, the language and especially the programming environment still provide a unique experience.

We have decided to do a new release because we have ported the virtual machine to the x86 architecture, so that it can run on the new Intel-based Macintosh computers (Mac Mini, MacBook,

iMac). The system is far from polished, but we have used Self on Mac OS X to do many hours of work on G4 Powerbooks and on the new Intel-based Macs.

Although our code is completely independent of theirs, we would be remiss if we did not mention Gordon Cichon and Harald Giese, who have also done an x86 port of Self. Their port runs on both Linux and Windows (which ours does not, yet - we would be thrilled if some kind soul were to port this latest version of Self to either of those platforms).

We hope that you will enjoy the chance to experience a different form of object-oriented programming.

Support. If you want to discuss Self with other interested people, there is a mailing list at self-interest@egroups.com. We would like to thank Jecel Assumpao Jr. for investing the time and effort to deeply understand the Self system, and furthermore for his help in explaining Self to many folks on the Self mailing list. Jecel also hosts the Self Swiki.

For information on the programming environment (essentially unchanged for Self 4.0), please refer to the Web page on Self 4.0.

Supplemental Information.

- An HTML version of the Self tutorial, 'Prototype-Based Application Construction Using Self 4.0', courtesy of Steve Dekorte. Thanks, Steve!
- In addition, see the Self bibliography for a listing of Self papers with on-line abstract

Acknowledgments.

This release was prepared by Alex Ausch, Adam Spitz and David Ungar. Kristen McIntyre helped with the PowerPC Mac port. Self owes its existence to the members of the Self Group and to the support of Sun Microsystems Laboratories, directed by Bob Sproull.

Sun, Sun Microsystems, the Sun Logo, Java, and HotSpot are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

12. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the United States and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

References

- | | | | |
|--------|--|---------|---|
| AC96 | Martin Abadi and Luca Cardelli. <i>A Theory of Primitive Objects: Untyped and First-Order Systems</i> . Information and Computation, 125(2):78-102, 15 March 1996. | AM95 | Malcolm Atkinson and Ronald Morrison. <i>Orthogonally Persistent Object Systems</i> . The International Journal on Very Large Data Bases, Vol. 4, Issue 3, July 1995. |
| AFGH04 | Matthew Arnold, Stephen J. Fink, David Grove, Michael Hind, and Peter F. Sweeney. <i>A Survey of Adaptive Optimization in Virtual Machines</i> . IBM Research Report RC23143 (W032-097), May 18, 2004. | APS93 | Ole Ageson, Jens Palsberg, and Michael I. Schwartzbach. <i>Type Inference of Self: Analysis of Objects with Dynamic and Multiple Inheritance</i> . Proc. ECOOP '93, pp. 247-267. Kaiserslautern, Germany, July 1993. |
| Age95 | Ole Ageson. <i>The Cartesian Product Algorithm</i> . ECOOP'95. | Assu | Jecel Assumpao Jr. <i>The Merlin Papers</i> . http://www.merlinterc.com/lsi/mpapers.html |
| Age96 | Ole Ageson. <i>Concrete Type Inference: Delivering Object-Oriented Applications</i> . PhD Dissertation, Computer Science, Stanford University, 1996. | AU94 | Ole Ageson and David Ungar. <i>Sifting Out the Gold: Delivering Compact Applications From an Exploratory Object-Oriented Environment</i> , OOPSLA'94. |
| Age97 | Ole Ageson. <i>The Design and Implementation of Pep, A JavaTM Just-In-Time Translator</i> . Theory and Practice of Object Systems, 3(2), 1997, pp. 127-155. | Bair | Anselm Baird-Smith. <i>OScheme Overview</i> . http://koala.ilog.fr/abaird/oscheme/oscheme.html . |
| AH95 | Ole Ageson and Urs Hözle. <i>Type Feedback vs. Concrete Type Inference: A Comparison of Optimization Techniques for Object-Oriented Languages</i> . OOPSLA'95. | BCFG99 | Michael G. Burke, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio J. Serrano, V. C. Sreedhar, Harini Srinivasan, and John Whaley. <i>The Jalapeño Dynamic Optimizing Compiler for Java</i> . Proceedings ACM 1999 Java Grande Conference. |
| | | BCR03 | Patrick Baudisch, Edward Cutrell, and George Robertson. <i>High-Density Cursor: A Visualization Technique that Helps Users Keep Track of Fast-Moving Mouse Cursors</i> . INTERACT'03, IOS Press (c) IFIP, 2003, pp. 236-243. |
| | | BDB00 | Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. <i>DYNAMO: A Transparent Dynamic Optimization System</i> . PLDI, June 2000, pp. 1-12. |
| | | BD81 | A. Borning and R. Duisberg. <i>Constraint-Based Tools for Building User Interfaces</i> . ACM Transactions on Graphics 5(4) pp. 345-374 (October 1981). |
| | | Beau06 | Gwenolé Beauchesne. The Official SheepShaver Home Page. http://sheepshaver.cebix.net . 2006. |
| | | BG93 | Gilad Bracha, David Griswold. <i>Strongtalk: Typechecking Smalltalk in a Production Environment</i> . OOPSLA'93. |
| | | BHJ86 | Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy. <i>Object Structure in the Emerald System</i> . OOPSLA'86. |
| | | Blas91 | G. Blaschek. <i>Type-Safe OOP with Prototypes: The Concept of Omega</i> , Structured Programming 12 (12) (1991) 1-9. |
| | | Blas94 | G. Blaschek. <i>Object-Oriented Programming with Prototypes</i> . Springer-Verlag, New York, Berlin 1994. |
| | | Blom | Ranier Blome. http://www.pasteur.fr/~letondal/object-based.html , Object-based PLs. |
| | | Broo | Frederick P. Brooks. <i>The Mythical Man-Month: Essays on Software Engineering, 20th Anniversary Edition</i> . Addison-Wesley Professional, Boston 1995. |
| | | BSUH87 | William Bush, A. Dain Samples, David Ungar, and Paul Hilfinger. <i>Compiling Smalltalk-80 to a RISC</i> . SIGPLAN Notices Vol. 22, Issue 10, 1987. Also in SIGARCH Computer Architecture News 15(5), 1987, ASPLOS'87, and SIGOPS Operating Systems Review 21(4), 1987. |
| | | BU04 | Gilad Bracha and David Ungar. <i>Mirrors: Design Principles for Meta-Level Facilities of Object-Oriented Programming Languages</i> , OOPSLA, 2004. |
| | | Card88 | Luca Cardelli. <i>A Semantics of Multiple Inheritance</i> . Information and Computation 76, pp. 138-164, 1988. |
| | | Card95 | Luca Cardelli. <i>A Language with Distributed Scope</i> , Computing Systems, 1995. |
| | | Cham | Craig Chambers and the Cecil Group. The Cecil Language: Specification and Rationale. http://www.cs.washington.edu/research/projects/cecil/www/pubs/cecil-spec.html . |
| | | Cham92 | Craig Chambers. <i>The Design and Implementation of the Self Compiler, an Optimizing Compiler for Object-Oriented Programming Languages</i> . PhD Thesis, Computer Science Department, Stanford University, April 1992. |
| | | Cham92a | Craig Chambers. <i>Object-Oriented Multi-Methods in Cecil</i> . ECOOP 1992. |
| | | Chan95 | Bay-Wei Chang. <i>Seity: Object-Focused Interaction in the Self User Interface</i> . PhD dissertation, Stanford University, 1995. |

CPGB94	Matthew Conway, Randy Pausch, Rich Gossweile, and Tommy Burnette. <i>Alice: A Rapid Prototyping System for Building Virtual Environments</i> . CHI'94 Conference Companion.	DS84	L. Peter Deutsch and Allan M. Schiffman. <i>Efficient Implementation of the Smalltalk-80 System</i> . In Proceedings of the 11th Annual ACM Symposium on the Principles of Programming Languages, pp. 297-302, Salt Lake City, UT, 1984.
CU89	Craig Chambers and David Ungar. <i>Customization: Optimizing Compiler Technology for Self, A Dynamically-Typed Object-Oriented Programming Language</i> . PLDI 1989. Also in <i>20 Years of the ACM/SIGPLAN Conference on Programming Language Design and Implementation (1979-1999): A Selection</i> , 2003.	Erns99	Erik Ernst. <i>gbeta, a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance</i> . PhD thesis, University of Arhus, Denmark, 1999.
CU90	Craig Chambers and David Ungar. <i>Iterative Type Analysis and Extended Message Splitting: Optimizing Dynamically-Typed Object-Oriented Programs</i> . In Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation, White Plains, NY, June, 1990. Published as SIGPLAN Notices 25(6), June, 1990.	Ernst	Erik Ernst. <i>gbeta</i> . http://www.daimi.au.dk/~ernst/gbeta/
CU90a	Bay-Wei Changs and David Ungar. <i>Experiencing Self Objects: An Object-Based Artificial Reality</i> . The Self Papers, Computer Systems Laboratory, Stanford University, 1990.	FG93	W. Finzer and L. Gould. <i>Rehearsal World: Programming by Rehearsal</i> . In W. Finzer and L. Gould. <i>Watch What I Do: Programming by Demonstration</i> . MIT Press, 1993, pp 79-100.
CU91	Craig Chambers and David Ungar. <i>Making Pure Object-Oriented Languages Practical</i> . In OOPSLA '91 Conference Proceedings, pp. 1-15, Phoenix, AZ, October, 1991.	FHM94	Kathleen Fisher, Furio Honsell, and John C. Mitchell. <i>A Lambda Calculus of Objects and Method Specialization</i> . Nordic Journal of Computing archive Volume 1, Issue 1, Pages: 3 - 37, 1994.
CU93	Bay-Wei Chang, David Ungar. <i>Animation from Cartoons to the User Interface</i> . Sun Microsystems Laboratories TR95-33. Also in UIST, 1993.	FQ03	Stephen J. Fink and Feng Qian. <i>Design, Implementation and Evaluation of Adaptive Recompilation with On-Stack Replacement</i> . International Symposium on Code Generation and Optimization (CGO'03), 2003
UCHH91	Craig Chambers, David Ungar, Bay-Wei Chang, and Urs Hözle. <i>Parents Are Shared Parts of Objects: Inheritance and Encapsulation in Self</i> . Journal of Lisp and Symbolic Computation, 4(3), Kluwer Academic Publishers, June, 1991.	FS02	D. Flanagan and D. Shafer. <i>JavaScript: The Definitive Guide</i> . O'Reilly, 2002.
CUL89	Craig Chambers, David Ungar, and Elgin Lee. <i>An Efficient Implementation of Self, a Dynamically-Typed Object-Oriented Language Based on Prototypes</i> . In OOPSLA '89 Conference Proceedings, pp. 49-70, New Orleans, LA, 1989. Published as SIGPLAN Notices 24(10), October, 1989.	GBO99	T.R.G. Green, A. Birning, T. O'Shea, M. Minoughan, and R.B. Smith. <i>The Stripetalk Papers: Understandability as a Language Design Issue in Object-Oriented Programming Systems in Prototype-Based Programming: Concepts, Languages and Applications</i> . Noble, J., Taivalsaari, A., Moore, I., (eds), Springer (1999) pp. 47-62.
CUS95	Bay-Wei Chang, David Ungar, and Randall B. Smith. <i>Getting Close to Objects</i> . In Burnett, M., Goldberg, A., and Lewis, T., editors, <i>Visual Object-Oriented Programming, Concepts and Environments</i> , pp. 185-198, Manning Publications, Greenwich, CT, 1995.	GR83	Adelai Goldberg and David Robson, <i>Smalltalk-80: The Language and Its Implementation</i> . Addison-Wesley, Reading, MA, 1983.
DB04	Jessie Deidecker and Dr. Werner Van Belle. <i>Actors in an Ad-Hoc Wireless Network Environment</i> . 2004.	GroF	Group F: Avri Bilovich, Chris Budd, Graham Cartledge, Rhys Evans, Mesar Hameed, and Michael Parry. http://www.bath.ac.uk/~ma3mp/SelfHome.htm .
Deid	Jessie Deidecker. <i>Ambient Oriented Programming</i> . http://prog.vub.ac.be/amop/research/papers.html .	GSO91	W.W. Gaver, R.B. Smith., and T. O'Shea. <i>Effective Sounds in Complex Systems: The ARKola Simulation</i> . Proc. CHI '91 conference, pp 85-90.
Dek06	Steve Dekorte. IO. http://www.iolangauge.com/	HCU91	Urs Hözle, Craig Chambers, and David Ungar. <i>Optimizing Dynamically-Typed Object-Oriented Programs using Polymorphic Inline Caches</i> . In ECOOP '91 Conference Proceedings, pp. 21-38, Geneva, Switzerland, July, 1991.
Deu83	L. Peter Deutsch. <i>The Dorado Smalltalk-80 Implementation: Hardware Architecture's Impact on Software Architecture</i> . In <i>Smalltalk-80: Bits of History, Words of Advice</i> , Glenn Krasner, ed. Addison-Wesley, 1983.	HCU92	Urs Hözle, Craig Chambers, and David Ungar. <i>Debugging Optimized Code with Dynamic Deoptimization</i> , in Proc. ACM SIGPLAN '92 Conferences on Programming Language Design and Implementation, pp. 32-43, San Francisco, CA (June 1992).
Deu88	L. Peter Deutsch. Richards benchmark. Personal communication, 1988.	HKKH96	H. Ossher, M. Kaplan, A. Katz, W. Harrison, and V. Kruskal, <i>Specifying Subject-Oriented Composition</i> , Theory and Practice of Object Systems, volume 2, number 3, 1996, Wiley & Sons.
DGBJ03	James C. Dehnert, Brian K. Grant, John P. Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. <i>The Transmeta Code Morphing Software: Using Speculation, Recovery, and Adaptive Retranslation</i> . Proceedings of the First Annual IEEE/ACM International Symposium on Code Generation and Optimization March 2003, San Francisco, California. http://doi.ieeecomputersociety.org/10.1109/CGO.2003.1191529 .	HN88	J. Hennessy and P. Nye. <i>Stanford Integer Benchmarks</i> . Stanford University, 1988.
DMC92	C. Dony, J. Malenfant, and P. Cointe, <i>Prototype-Based Languages: From a New Taxonomy to Constructive Proposals and Their Validation</i> . In Proc. OOPSLA'92, pp. 201-217.	Hoar73	C. A. R. Hoare. <i>Hints on Programming Language Design</i> . SIGACT/SIGPLAN Symposium on Principles of Programming Languages, October 1973. First published as Stanford University Computer Science Dept. Technical Report No. CS-73-403, Dec. 1973.
DN66	Ole-Johan Dahl and Kristen Nygaard. <i>SIMULA—an ALGOL-Based Simulation Language</i> . CACM 9(9), Sept. 1966, pp. 671-678.	Höl94	Urs Hözle. <i>Adaptive Optimization for Self: Reconciling High Performance with Exploratory Programming</i> . PhD Thesis, Stanford University, Computer Science Department, 1994.
Doe03	Osvaldo Pinali Doederlein. <i>The Tale of Java Performance</i> . Journal of Object Technology, Vol. 2, No. 5, Sept.-Oct. 2003. http://www.jot.fm/issues/issue_2003_09/column3 .	HU94	Urs Hözle and David Ungar. <i>Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback</i> . In Proceedings of the SIGPLAN 94 Conference on Programming Language Design and Implementation, Orlando, FL, June, 1994.
		HU94a	Urs Hözle and David Ungar. <i>A Third Generation Self Implementation: Reconciling Responsiveness with Performance</i> . In OOPSLA'94 Conference Proceedings, pp. 229-243, Portland, OR, October, 1994. Published as SIGPLAN Notices 29(10), October, 1994.

HU95	Urs Hözle and David Ungar. <i>Do Object-Oriented Languages Need Special Hardware Support?</i> ECOOP'95.	MGD90	Brad A. Myers, Dario Giuse, Roger B. Dannenberg, Brad Vander Zanden, David Kosbie, Ed Pervin, Andrew Mickish, and Philippe Marchal. <i>Comprehensive Support for Graphical, Highly-Interactive User Interfaces: The Garnet User Interface Development Environment</i> . This article was printed as <i>Garnet: Comprehensive Support for Graphical, Highly-Interactive User Interfaces</i> , IEEE Computer. Vol. 23, No. 11. November, 1990. pp. 71-85. Translated into Japanese and reprinted in Nikkei Electronics, No. 522, March 18, 1991, pp. 187-205.
HU96	Urs Hözle and David Ungar. <i>Reconciling Responsiveness with Performance in Pure Object-Oriented Languages</i> . TOPLAS. 1996.		
D'Hont	Theo D'Hont. <i>About PROG</i> . http://prog.vub.ac.be/progsite .		
Inga81	Daniel H. H. Ingalls. <i>Design Principles Behind Smalltalk</i> . BYTE Magazine, August 1981, pp. 286-298.		
Iver79	Kenneth E. Iverson. <i>Operators</i> . ACM TOPLAS, Vol. 1 No. 2, October 1979, pp. 161-176.	MMM97	Brad A. Myers, Richard G. McDaniel, Robert C. Miller, Alan S. Ferrenny, Andrew Faulring, Bruce D. Kyle, Andrew Mickish, Alex Klimovitski, and Patrick Doane. <i>The Amulet Environment: New Models for Effective User Interface Software Development</i> . IEEE Transactions on Software Engineering, Vol. 23, no. 6. June, 1997. pp. 347-365.
John79	Ronald L. Johnston. <i>The Dynamic Incremental Compiler of APL3000</i> . Proceedings of the APL'79 Conference. Published as APL Quote Quad 9(4), p. 82-87, 1979.	MS95	John Maloney and Randall B. Smith, <i>Directness and Liveness in the Morphic User Interface Construction Environment</i> . UIST'95.
John88	Ralph E. Johnson, Justin O. Graver, and Lawrence W. Zurawski. TS: An Optimizing Compiler for Smalltalk. In <i>OOPSLA'88 Conference Proceedings</i> , pp. 18-26, San Diego, CA, 1988. Published as <i>SIGPLAN Notices</i> 23(11), November, 1988.	Mule95	Phillipe Mulet, <i>Réflexion & Langages à Prototypes</i> , PhD Dissertation, Ecole des Mines de Nantes, France, 1995.
Jone89	Charles M. Jones. <i>Chuck Amuck: The Life and Times of an Animated Cartoonist</i> . Farrar Straus Gifoux, New York, 1989.	Myer97	Brad Myers. <i>Amulet Overview</i> . http://www.cs.cmu.edu/afs/cs/project/amulet/www/amulet-overview.html
Kay93	Alan Kay. <i>The Early History of Smalltalk</i> . ACM SIGPLAN Notices Vol. 28, Issue 3, March 1993. Also HOPL-II, 1993.	Pal90	Joseph Pallas. <i>Multiprocessor Smalltalk: Implementation, Performance, and Analysis</i> . PhD Dissertation. Stanford University, 1990.
Krin03	Chandra Krintz. <i>Coupling On-Line and Off-Line Profile Information to Improve Program Performance</i> . CGO'03.	Parn72	D. L. Parnas. <i>On the Criteria To Be Used in Decomposing Systems into Modules</i> . CACM, Vol. 15, No. 12, December 1972. pp. 1053-1058.
Lee88	Elgin Lee. <i>Object Storage and Inheritance for Self</i> . Engineer's thesis, Electrical Engineering Department, Stanford University, 1988.	PKB86	J. M. Pendleton, S.I. Kong, E.W. Brown, F. Dunlap, C. Marino, D. M. Ungar, D.A. Patterson, and D. A. Hodges. <i>A 32-bit Microprocessor for Smalltalk</i> . IEEE Journal of Solid-State Circuits, Vol. 21, Issue 5, Oct. 1986, pp. 741-749.
Len05	Mark Lentczner. <i>Welcome to Wheat</i> . www.wheatfarm.org	PMH05	Ellen Van Paesschen, Wolfgang De Meuter, and Maja D'Hondt. <i>SelfSync: a Dynamic Round-Trip Engineering Environment</i> . In <i>OOPSLA'05: Companion of the 20th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications</i> . San Diego, U.S.A. ACM Press, 2005. http://prog.vub.ac.be/Publications/2005/vub-prog-tr-05-20.pdf .
Lieb86	Henry Lieberman. <i>Using Prototypical Objects to Implement Shared Behavior in Object Oriented System</i> . OOPSLA'86.	RB97	Don Roberts, John Brant, and Ralph Johnson. <i>A Refactoring Tool for Smalltalk</i> . Theory and Practice of Object Systems, Vol. 3, Issue 4, pp. 253-263, 1997, John Wiley & Sons, Inc.
Ling04	K Lyngfelt, <i>MorphR—A Morphic GUI in Ruby</i> . Masters Thesis, University of Trollhättan ÅE Uddevalla., 2004.	RMC91	George Robertson, Jock Mackinlay, and Stuart Card. <i>Cone Trees: Animated 3D Visualizations of Hierarchical Information</i> . ACM Computer Human Interface Conference, 1991.
LJ87	G. Lakoff and M. Johnson. <i>Women, Fire, and Dangerous Things: What Categories Reveal About the Mind</i> . University of Chicago Press, 1987.	RRS99	Ian Rogers, Alasdair Rawsthorne, Jason Sologlou. <i>Exploiting Hardware Resources: Register Assignment Across Method Boundaries</i> . First Workshop on Hardware Support for Objects and Microarchitectures for Java, 1999.
LTP86	WR LaLonde, DA Thomas, JR Pugh, <i>An Exemplar Based Smalltalk</i> . OOPSLA'86.	RS	Brian T. Rice and Lee Salzman. <i>The Home of the Slate Programming Language</i> . http://slate.tunes.org/
Mal01	John Maloney. <i>An Introduction to Morphic: The Squeak User Interface Framework</i> . In <i>Squeak: Open Personal Computing and Multimedia</i> , Mark Guzdial and Kim Rose (eds.), Prentice Hall, 2001.	SA05	Lee Salzman and Johnathan Aldrich. <i>Prototypes with Multiple Dispatch: An Expressive and Dynamic Object Model</i> . ECOOP 2005.
MMM90	Ole Lehrmann Madsen, Boris Magnusson, and Birger Møller-Pedersen. <i>Strong Typing of Object-Oriented Languages Revisited</i> . In ECOOP/OOPSLA'90 Conference Proceedings, pp. 140-149, Ottawa, Canada, October, 1990.	SC02	Benoit Sonntag and Dominique Colnet. <i>Lisaac: The Power of Simplicity at Work for Operating System</i> . ACM Fortieth International Conference on Tools Pacific: Objects for internet, mobile, and embedded applications, 2002.
MMN93	Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. <i>Object-Oriented Programming in the Beta Programming Language</i> . Addison-Wesley Publishing Co., Wokingham, England, 1993.	SDNB03	Nathaneal Scharli, Stephane Ducasse, Oscar Nierstrasz, and Andrew Black. <i>Traits: Composable Units of Behavior</i> . ECOOP 2003.
Moor	Ivan Moore. Guru. http://selfguru.sourceforge.net/index.html	Setal90	R.B. Smith, T. O'Shea, E. O'Malley, E. Scanlon, and J. Taylor. <i>Preliminary experiments with a distributed, multimedia problem solving environment</i> , in Proceedings of EC-CSCW '90, Gatwick, England, pp 19-34. Also appears in J. Bowers and S. Benford, eds., <i>Studies in Computer Supported Cooperative Work: theory, practice and design</i> , Elsevier, Amsterdam, (1991) pp 31-48.
Moo95	Ivan Moore. <i>Guru - A Tool for Automatic Restructuring of Self Inheritance Hierarchies</i> at TOOLS USA 1995		
Moo96	Ivan Moore. <i>Automatic Inheritance Hierarchy Restructuring and Method Refactoring</i> . OOPSLA'96.		
Moo96a	Ivan Moore. <i>Automatic Restructuring of Object-Oriented Programs</i> . PhD Thesis, Manchester University, 1996.		
MC93	P. Mulet and P. Cointe, <i>Definition of a Reflective Kernel for a Prototype-Based Language</i> . In Proceedings of the 1st International Symposium on Object Technologies for Advanced Software, Kanazawa, Japan, Springer-Verlag, Berlin. S. Nishio and A. Yonezawa (eds.), pp. 128-144, 1993. http://citeseer.ist.psu.edu/article/mulet93definition.html .		
MC96	Ivan Moore and Tim Clement. <i>A Simple and Efficient Algorithm for Inferring Inheritance Hierarchies</i> at TOOLS Europe 1996.		

Setal93	E. Scanlon, R.B. Smith, T. O'Shea, C. O'Malley, and J. Taylor. <i>Running in the Rain—Can a Shared Simulation Help To Decide?</i> Physics Education, Vol 28, March 1993, pp 107-113.	SU95	Randall Smith and David Ungar. <i>Programming as an Experience: The Inspiration for Self.</i> In Proceedings of the 9th European Conference, Århus, Denmark, Aug. 1995. Published as Lecture Notes in Computer Science 952: ECOOP'95 — Object-Oriented Programming, Walther Olthoff (ed.) Springer, Berlin. pp 303-330.
Setal99	R.B. Smith, M.J. Sipusic, and R.L. Pannoni. <i>Experiments Comparing Face-to-Face with Virtual Collaborative Learning.</i> Proceedings of Conference on Computer Support for Collaborative Learning 1999. Stanford University, Palo Alto, December 1999. pp 558-566.	SU96	Randall Smith and David Ungar. <i>A Simple and Unifying Approach to Subjective Objects,</i> in Theory and Practice of Object Systems, Vol. 2, Issue 3, Special Issue on subjectivity in object-oriented systems, 1996.
SL93	B. Schwartz and M. Lentczner. <i>Direct Programming Using a Unified Object Model.</i> In OOPSLA'92 Addendum to the Proceedings. Published as OOPS Messenger, 4.2, (1993) 237.	SUC92	Randall B. Smith, David Ungar, and Bay-Wei Chang, <i>The Use Mention Perspective on Programming for the Interface,</i> In Brad A. Myers, <i>Languages for Developing User Interfaces,</i> Jones and Bartlett, Boston, MA, 1992. pp 79-89.
SLST94	R. B. Smith, M. Lentczner, W. Smith, A. Taivalsaari, and D. Ungar, <i>Prototype-Based Languages: Object Lessons from Class-Free Programming (Panel),</i> in Proc. OOPSLA'94, pp. 102-112 (October 1994). Also see the panel summary of the same title, in Addendum to the Proceedings of OOPSLA '94, pp. 48-53.	Suth63	Ivan Sutherland. <i>Sketch Pad,</i> AFIPS Spring Joint Computer Conference, Detroit, 1963.
SLU88	Lynn Stein, Henry Lieberman, and David Ungar. <i>The Treaty of Orlando.</i> ACM SIGPLAN Notices, Vol. 23, Issue 5, May 1988. Also in W. Kim and F. Lochovsky (eds.), <i>Object-Oriented Concepts, Databases, and Applications.</i> ACM Press and Addison-Wesley.	SWU97	Randall B. Smith, Mario Wolczko, and David Ungar. <i>From Kansas to Oz: Collaborative Debugging When a Shared World Breaks.</i> CACM, April, 1997. pp 72-78.
Smi87	Randall B. Smith. <i>Experiences with the Alternate Reality Kit, an Example of the Tension Between Literalism and Magic.</i> Proc. CHI + GI Conference, pp 61-67, Toronto, (April 1987).	Tai92	Antero Taivalsaari. <i>Kevo - A Prototype-Based Object-Oriented Language Based on Concatenation and Module Operations.</i> University of Victoria Technical Report DCS-197-1R, Victoria, B.C., Canada, June 1992.
Smi91	Smith, R.B., <i>A Prototype Futuristic Technology for Distance Education,</i> in New Directions in Educational Technology, Scanlon, E., and O'Shea, T., (eds.) Springer, Berlin, (1991) pp 131-138.	Tai93	Antero Taivalsaari. <i>A Critical View of Inheritance and Reusability in Object-Oriented Programming.</i> PhD dissertation, Jyväskylä Studies in Computer Science, Economics and Statistics 23, University of Jyväskylä, Finland, December 1993, 276 pages (ISBN 951-34-0161-8).
Smi93	D. Smith. <i>Pygmalion: An Executable Electronic Blackboard.</i> In W. Finzer and L. Gould, <i>Watch What I Do: Programming by Demonstration,</i> MIT Press, 1993, pp 19-48.	Tai93a	Antero Taivalsaa, <i>Concatenation-Based Object-Oriented Programming in Kevo,</i> Actes de la 2eme Conference sur la Representations Par Objets RPO'93 (La Grande Motte, France, June 17-18, 1993), Published by EC2, France, June 1993, pp.117-130.
Smi94	Walter Smith. <i>Self and the Origins of NewtonScript.</i> PIE Developers Magazine, July 1994. http://wsmith.best.vwh.net/Self/intro.html .	TC95	Bruce H. Thomas and Paul Calder. <i>Animating Direct Manipulation Interfaces.</i> UIST'95.
Smi95	Walter R. Smith. <i>Using a Prototype-based Language for User Interface: The Newton Project's Experience.</i> OOPSLA'95.	TC01	Bruce H. Thomas and Paul Calder. <i>Applying Cartoon Animation Techniques to Graphical User Interfaces.</i> ACM Transactions on Computer-Human Interaction, Vol. 8, No. 3, September 2001, Pages 198–222.
SMU95	Randall B. Smith, John Maloney, and David Ungar, <i>The Self-4.0 User Interface: Manifesting a System-wide Vision of Concreteness, Uniformity, and Flexibility.</i> OOPSLA '95.	TD02	B. H. Thomas and V. Demczuk. <i>Which Animation Effects Improve Indirect Manipulation?</i> Interacting with Computers 14, 2002, pp. 211-229. Elsevier. www.elsevier.com/locate/intcom
SOSL97	E. Scanlon, T. O'Shea, R.B. Smith, and Y. Li. <i>Supporting the Distributed Synchronous Learning of Probability: Learning from an Experiment,</i> In R. Hall, N. Miyake, and N. Enedy (eds.) Proceedings of CSCL'97, The Second International Conference on Computer Support for Collaborative Learning Toronto, December 10-14, 1997, pp 224-230.	Tha86	Chuck Thacker. <i>Personal Distributed Computing: The Alto and Ethernet Hardware.</i> ACM Conference on the History of Personal Workstations, 1986.
SS79	Guy Steele, Jr. and Gerald Jay Sussman. <i>Design of LISP-based Processors, or SCHEME: A Dielectric LISP, or Finite Memories Considered Harmful, or LAMBDA: The Ultimate Opcode.</i> AI Memo No. 514, MIT AI Laboratory, March 1979.	Thom98	Bruce H. Thomas. <i>Warping to Enhance 3D User Interfaces.</i> APCHI, p. 169, Third Asian Pacific Computer and Human Interaction, 1998. http://doi.ieeecomputersociety.org/10.1109/APCHI.1998.704188
SSP99	M.J. Sipusic, R.L. Pannoni, R.B. Smith, J. Dutra, J.F. Gibbons, and W.R. Sutherland. <i>Virtual Collaborative Learning: A Comparison between Face-to-Face Tutored Video Instruction (TVI) and Distributed Tutored Video Instruction (DTVI),</i> Sun Laboratories Technical Report SMLI-TR-99-72.	TJ84	Frank Thomas and Ollie Johnson. <i>Disney Animation: The Illusion of Life.</i> New York, Abbeville, 1984.
Strou86	Bjarne Stroustrup. <i>An Overview of C++.</i> SIGPLAN Notices, Vol. 21, #10, October 1986, pp. 7-18	TK02	Robert Tolksdorf and Kai Knubben. <i>Programming Distributed Systems with the Delegation-Based Object-Oriented Language dSelf.</i> ACM Symposium on Applied Computing, 2002, pp. 927-931.
StV96	<i>Self: the Video.</i> Videotape by Sun Microsystems Laboratories, Sun Labs document #0448, Oct. 1996.	TSS88	C. P. Thacker, L. Stewart, and E. H. Satterthwaite. <i>Firefly: A Multiprocessor Workstation.</i> IEEE Transactions on Computers, 37(8):909-920, Aug. 1988.
SU94	R.B. Smith and D. Ungar. <i>Us: A Subjective Language with Perspective Objects.</i> SML-94-0416.	Ung84	David Ungar. <i>Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm.</i> ACM Symposium on Practical Software Development Environments, 1984. Also in SIGPLAN Notices Vol. 19, Issue 5, 1984 and SIGSOFT Software Engineering Notices Vol. 9, Issue 3, 1984.
		Ung87	David Ungar. <i>The Design and Evaluation of a High-Performance Smalltalk System.</i> MIT Press, 1987.
		Ung95	David Ungar. <i>Annotating Objects for Transport to Other Worlds.</i> OOPSLA'95.

- | | | | |
|--------|--|-------|--|
| US87 | David Ungar and Randall B. Smith, <i>Self: The Power of Simplicity</i> , Proceedings of the 1987 ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA), Orlando, FL, October, 1987, pp. 227–242. A revised version appeared in the Journal of Lisp and Symbolic Computation, 4(3), Kluwer Academic Publishers, June, 1991. | WAU96 | Mario Wolczko, Ole Agesen, and David Ungar. <i>Towards a Universal Implementation Substrate for Object-Oriented Languages</i> . Sun Labs 96-0506. http://www.sunlabs.com/people/mario/pubs/substrate.pdf , 1996. Also presented at OOPSLA'99 workshop on Simplicity, Performance and Portability in Virtual Machine Design. |
| USCH92 | David Ungar, Randall B. Smith, Craig Chambers, and Urs Hözle. <i>Object, Message, and Performance: How They Coexist in Self</i> . Computer, 25(10), pp. 53-64. (October 1992). | Wha01 | John Whaley. <i>Partial method compilation using dynamic profile information</i> . OOPSLA'01. |
| USA05 | David Ungar, Adam Spitz, and Alex Ausch. <i>Constructing a Metacircular Virtual Machine in an Exploratory Programming Environment</i> . OOPSLA 2005. | Wol96 | Mario Wolczko. <i>Self Includes: Smalltalk</i> . In <i>Prototype-Based Programming: Concepts, Languages and Applications</i> , J. Noble, A. Taivalsaari, and I. Moore, (eds), Springer (1999). |

The When, Why and Why Not of the BETA Programming Language

Bent Bruun Kristensen
University of Southern Denmark
Campusvej 55
DK-5230 Odense M, Denmark
+45 65 50 35 39
bbk@mmmi.sdu.dk

Ole Lehrmann Madsen
University of Aarhus
Åbogade 34
DK-8200 Århus N, Denmark
+45 89 42 56 70
ole.l.madsen@daimi.au.dk

Birger Møller-Pedersen
University of Oslo
Gaustadalleen 23
NO-0316 Oslo, Norway
+47 22 85 24 37
birger@ifi.uio.no

Abstract

This paper tells the story of the development of BETA: a programming language with just one abstraction mechanism, instead of one abstraction mechanism for each kind of program element (classes, types, procedures, functions, etc.). The paper explains how this single abstraction mechanism, the pattern, came about and how it was designed to be so powerful that it covered the other mechanisms.

In addition to describing the technical challenge of capturing all programming elements with just one abstraction mechanism, the paper also explains how the language was based upon a modeling approach, so that it could be used for analysis, design and implementation. It also illustrates how this modeling approach guided and settled the design of specific language concepts.

The paper compares the BETA programming language with other languages and explains how such a minimal language can still support modeling, even though it does not have some of the language mechanisms found in other object-oriented languages.

Finally, the paper tries to convey the organization, working conditions and social life around the BETA project, which turned out to be a lifelong activity for Kristen Nygaard, the authors of this paper, and many others.

Categories and subject descriptors: D.3.2 [PROGRAMMING LANGUAGES]: Language Classifications – BETA; D.3.3 [PROGRAMMING LANGUAGES]: Language Constructs and Features; K.2 [HISTORY OF COMPUTING] Software; D.1.5 [PROGRAMMING TECHNIQUES]: Object-oriented Programming; **General Terms:** Languages; **Keywords:** programming languages, object-oriented programming, object-oriented analysis, object-oriented design, object-oriented modeling, history of programming.

Permission to make digital/hard copy of part of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date of appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Permission may be requested from the Publications Dept., ACM, Inc., 2 Penn Plaza, New York, NY 11201-0701, USA, fax:+1(212) 869-0481, permissions@acm.org

©2007 ACM 978-1-59593-766-7/2007/06-ART10 \$5.00

DOI 10.1145/1238844.1238854

<http://doi.acm.org/10.1145/1238844.1238854>

1. Introduction

This paper is a description of what BETA is, why it became what it is and why it lacks some of the language constructs found in other languages. In addition, it is a history of the design and implementation of BETA, its main uses and its main influences on later research and language efforts.

BETA is a programming language that has only one abstraction mechanism, the pattern, covering abstractions like record types, classes with methods, types with operations, methods, and functions. Specialization applies to patterns in general, thus providing a class/subclass mechanism for class patterns, a subtype mechanism for type patterns, and a specialization mechanism for methods and functions. The latter implies that inheritance is supported for methods – another novel characteristic of BETA. A pattern may be virtual, providing virtual methods as in other object-oriented languages. Since a pattern may be used as a class, virtuality also supports virtual classes (and types).

This paper is also a contribution to the story of the late Kristen Nygaard, one the pioneers of computer science, or informatics as he preferred to call it. Nygaard started the BETA project as a continuation of his work on SIMULA and system description. This was the start of a 25-year period of working with the authors, not only on the design of the BETA language, but also on many other aspects of informatics.

The BETA project was started in 1976 and was originally supposed to be completed in a year or two. For many reasons, it evolved into an almost lifelong activity involving Nygaard, the authors of this paper and many others. The BETA project became an endeavor for discussing issues related to programming languages, programming and informatics in general.

The BETA project covers many different kinds of activities from 1976 until today. We originally tried to write this paper in historic sequence, and so that it can be read with little or no prior knowledge of BETA. We have not, however, organized the paper according to time periods, since the result included a messy mix of distinct types of

events and aspects, too much overlap, and too little focus on important aspects. The resulting paper is organized as follows:

- Section 2 describes the background of the project.
- Section 3 describes the course of the BETA project, including people, initial research ideas, project organization and the process as well as personal interactions.
- Section 4 describes the motivation and development of the modeling aspects and the conceptual framework of BETA.
- Section 5 describes parts of the rationale for the BETA language, the development of the language, and essential elements of BETA.
- Section 6 describes the implementation of BETA.
- Section 7 describes the impact and further development of BETA.

Sections 3-6 form the actual story of BETA enclosed by background (section 2) and impact (section 7). Sections 3-6 describe distinct aspects of BETA. The story of the overall BETA project in section 3 forms the foundation/context for the following aspects. This is how it all happened. Modeling is essential for the design of BETA. This perspective on design of and programming in object-oriented languages is presented in section 4. Throughout the presentation of the various language elements in the following section the choices are discussed and motivated by the conceptual framework in section 4. Section 5 presents the major elements of BETA. Because BETA may be less known, a more comprehensive presentation is necessary in order to describe its characteristics. However, the presentation is still, and should be, far from a complete definition of the language. Finally, the implementation of BETA, historically mainly following after the language design, is outlined in section 6.

In order to give a sequential ordering of the various events and activities, a timeline for the whole project is shown in the appendix. In the text events shown in the timeline are printed in Tunga font. For example, text like: "... BETA Project start ..." means that this is an event shown in the time line.

2. Background

This section describes the background and setting for the early history of BETA. It includes personal backgrounds and a description of the important projects leading to the BETA project.

2.1 People

The BETA project was started in 1976 at the Computer Science Department, Aarhus University (DAIMI). Bent Bruun Kristensen and Ole Lehrmann Madsen had been

students at DAIMI since 1969 – Birger Møller-Pedersen originally started at the University of Copenhagen, but moved to DAIMI in 1974. Nygaard was Research Director at the Norwegian Computing Centre (NCC), Oslo, where the SIMULA languages [32-34, 130] were developed in the sixties.

In the early seventies, the programming language scene was strongly influenced by Pascal [161] and structured programming. SIMULA was a respected language, but not in widespread use. Algol 60 [129] was used for teaching introductory programming at DAIMI. Kristensen and Madsen were supposed to be introduced to SIMULA as the second programming language in their studies. However, before that happened Pascal arrived on the scene in 1971, and most people were fascinated by its elegance and simplicity as compared to Algol. Pascal immediately replaced SIMULA as the second language and a few years later Pascal also replaced Algol as the introductory language for teaching at DAIMI. A few people, however, found SIMULA superior to Pascal: the Pascal record and variant record were poor substitutes for the SIMULA class and subclass.

Although SIMULA was not in widespread use, it had a strong influence on the notion of structured programming and abstract data types. The main features of SIMULA were described in the famous book by Dahl, Dijkstra and Hoare on structured programming [29]. Hoare's groundbreaking paper *Proof of Correctness of Data Representation* [56] introduced the idea of defining abstract data types using the SIMULA class construct and the notion of class invariant.

Kristen Nygaard visiting professor at DAIMI. Nygaard became a guest lecturer at DAIMI in 1973; in 1974/75 he was a full-time visiting professor, and after that he continued as a guest lecturer for several years. Among other things, Nygaard worked with trade unions in Norway to build up expertise in informatics. At that time there was a strong interest among many students at DAIMI and other places in the social impact of computers. Nygaard's work with trade unions was very inspiring for these students. During the '70s and '80s a number of similar projects were carried out in Scandinavia that eventually led to the formation of the research discipline of *system development with users*, later called *participatory design*. The current research groups at DAIMI in object-oriented software systems and human computer interaction are a direct result of the cooperation with Nygaard. This is, however, another story that will not be told here. The design of BETA has been heavily influenced by Nygaard's overall perspective on informatics including social impact, system description with users, philosophy, and programming languages. For this reason the story of BETA cannot be told without relating it to Nygaard's other activities.

Morten Kyng was one of the students at DAIMI who was interested in social aspects of computing. In 1973 he listened to a talk by Nygaard at the Institute of Psychology at Aarhus University. After the talk he told Nygaard that he was at the wrong place and invited him to repeat his talk at DAIMI. Kyng suggested to DAIMI that Nygaard be invited as a guest lecturer. The board of DAIMI decided to do so, since he was considered a good supplement to the many theoretical disciplines in the curriculum at DAIMI at that time. Madsen was a student representative on the board; he was mainly interested in compilers and was thrilled about Nygaard being a guest lecturer. He thought that DAIMI would then get a person that knew about the SIMULA compiler. This turned out not to be the case: compiler technology was not his field. This was our first indication that Nygaard had a quite different approach to informatics and language design from most other researchers.

2.2 The SIMULA languages

Since SIMULA had a major influence on BETA we briefly mention some of the highlights of SIMULA. A comprehensive history of the SIMULA languages may be found in the HOPL-I proceedings [35] and in [107]. SIMULA and object-oriented programming were developed by Ole-Johan Dahl and Nygaard. Nygaard's original field was operations research and he realized early on that computer simulations would be a useful tool in this field. He then made an alliance with Dahl, who – as Nygaard writes in an obituary for Dahl [132] – had an exceptional talent for programming. This unique collaboration led to the first SIMULA language, SIMULA I, which was a simulation language. Dahl and Nygaard quickly realized that the concepts in SIMULA I could be applied to programming in general and as a result they designed SIMULA 67 – later on just called SIMULA. SIMULA is a general-purpose programming language that contains Algol as a subset.

Users of today's object-oriented programming languages are often surprised that SIMULA contains many of the concepts that are now available in mainstream object-oriented languages:

- Class and object: A class defines a template for creating objects.
- Subclass: Classes may be organized in a classification hierarchy by means of subclasses.
- Virtual methods: A class may define virtual methods that can be redefined (sometimes called overridden) in subclasses.
- Active objects: An object in SIMULA is a coroutine and corresponds to a thread.
- Action combination: SIMULA has an “inner” construct for combining the statement-parts of a class and a subclass.

- Processes and schedulers: It is straightforward in SIMULA to write new concurrency abstractions including schedulers.
- Frameworks: SIMULA provided the first object-oriented framework in form of class Simulation, which provided SIMULA I's simulation features.
- Automatic memory management, including garbage collection.

Most of the above concepts are now available in object-oriented languages such as C++ [148], Eiffel [125], Java [46], and C# [51]. An exception is the SIMULA notion of an active object with its own action sequence, which strangely enough has not been adopted by many other languages (one exception is UML). For Dahl and Nygaard it was essential to be able to model concurrent processes from the real world.

The ideas of SIMULA have been adopted over a long period. Before object orientation caught on, SIMULA was very influential on the development of abstract data types. Conversely, ideas from abstract data types later led to an extension of SIMULA with constructs like **public**, **private** and **protected** – originally proposed by Jakob Palme [137].

2.3 The DELTA system description language

When Nygaard came to DAIMI, he was working on system description and the design of a new language for system description based on experience from SIMULA. It turned out that many users of SIMULA seemed to get more understanding of their problem domain by having to develop a model using SIMULA than from the actual simulation results. Nygaard together with Erik Holbæk-Hanssen and Petter Håndlykken had thus started a project on developing a successor to SIMULA with main focus on system description, rather than programming. This led to a language called DELTA [60].

DELTA means ‘participate’ in command form in Norwegian. The name indicates another main goal of the DELTA language. As mentioned, Nygaard had started to include users in the design of systems and DELTA was meant as a language that could also be used to communicate with users – DELTA (participate!) was meant as an encouragement for users to participate in the design process.

The goal of DELTA was to improve the SIMULA mechanisms for describing real-world systems. In the real world, activities take place concurrently, but real concurrency is not supported by SIMULA. To model concurrency SIMULA had support for so-called quasi-parallel systems. A simulation program is a so-called discrete event system where a simulation is driven by discrete events generated by the objects of the simulation. All state changes had to be described in a standard imperative way by remote procedure calls (message calls),

assignments and control structures. DELTA supports the description of true concurrent objects and uses predicates to express state changes and continuous changes over time. The use of predicates and continuous state changes implied that DELTA could not be executed, but as mentioned the emphasis was on system description.

DELTA may be characterized as a specification language, but the emphasis was quite different from most other specification languages at that time such as algebraic data types, VDL, etc. These other approaches had a mathematical focus in contrast to the system description (modeling) focus of DELTA.

DELTA had a goal similar to that of the object-oriented analysis and design (OOA/OOD) methodologies (like that of Coad and Yourdon [25]) that appeared subsequently in the mid-'80s. The intention was to develop languages and methodologies for modeling real-world phenomena and concepts based on object-oriented concepts. Since SIMULA, modeling has always been an inherent part of language design in the Scandinavian school of object orientation. The work on DELTA may be seen as an attempt to further develop the modeling capabilities of object-orientation.

The report describing DELTA is a comprehensive description of the language and issues related to system description. DELTA has been used in a few projects, but it is no longer being used or developed.

The system concept developed as part of the DELTA project had major influence on the modeling perspective of BETA – in Section 4.1 we describe the DELTA system concept as interpreted for BETA.

2.4 The Joint Language Project

BETA project start. The BETA project was started in 1976 as part of what was then called the Joint Language Project (JLP). The JLP was a joint project between researchers at DAIMI, The Regional Computing Center at the University of Aarhus (RECAU), the NCC and the University of Aalborg.

Joint Language Project start. The initiative for the JLP was taken in the autumn of 1975 by the late Bjarner Svejgaard, director of RECAU. Svejgaard suggested to Nygaard that it would be a good idea to define a new programming language based on the best ideas from SIMULA and Pascal. Nygaard immediately liked the idea, but he was more interested in a successor to SIMULA based on the ideas from DELTA. In the BETA Language Development report from November 1976 [89] the initial purpose of the JLP was formulated as twofold:

1. To develop and implement a high-level programming language as a projection of the DELTA system

description language into the environment of computing equipment.

2. To provide a common central activity to which a number of research efforts in various fields of informatics and at various institutions could be related.

The name GAMMA was used for this programming language.

JLP was a strange project: on the one hand there were many interesting discussions of language issues and problems, while on the other hand there was no direct outcome. At times we students on the project found it quite frustrating that there was no apparent progress. We imagine that this may have been frustrating for the other members of the project as well. In hindsight we believe that the reason for this may have been a combination of the very different backgrounds and interests of people in the team combined with Nygaard's lack of interest in project management. Nygaard's strengths were his ability to formulate and pursue ambitious research goals, and his approach to language design with emphasis on modeling was unique.

Many issues were discussed within the JLP, mainly related to language implementation and some unresolved questions about the DELTA language. As a result six subprojects were defined:

- **Distribution and maintenance.** This project was to discuss issues regarding software being deployed to a large number of computer installations of many different types. This included questions such as distribution formats, standardized updating procedures, documentation, interfaces to operating systems, etc.
- **Value types.** The distinction between object and value was important in SIMULA and remained important in DELTA and BETA. For Nygaard classes were for defining objects and types for defining values. He found the use of the class concept for defining abstract data types a ‘doubtful approach, easily leading to conceptual confusion’ with regard to objects and values. In this paper we use the term value type¹ when we refer to types defining values. The purpose of this subproject was to discuss the definition of value types. We return to value types in Sections 5.1.1 and 5.8.2.
- **Control structures** within objects. The purpose of this subproject was to develop the control structures for GAMMA.
- **Contexts.** The term “system classes” was used in SIMULA to denote classes defining a set of predefined concepts (classes) for a program. The classes SIMSET and SIMULATION are examples of such system classes.

¹ In other contexts, we use the term *type*, as is common within programming languages.

Møller-Pedersen later revised and extended the notion of system classes and proposed the term “context”. In today’s terminology, class SIMSET was an example of a class library providing linked lists and class SIMULATION was an example of a class framework (or application framework).

- **Representative states.** A major problem with concurrent programs was (and still is) to ensure that interaction between components results only in meaningful states of variables – denoted representative states in JLP. At that time, there was much research in concurrent programming including synchronization, critical regions, monitors, and communication. This subproject was to develop a conceptual approach to this problem, based upon the concepts of DELTA and work by Lars Mathiassen and Morten Kyng.

- **Implementation language.** In the early seventies, it was common to distinguish between general programming languages and implementation languages. An implementation language was often defined as an extended subset of the corresponding programming language. The subset was supposed to contain the parts that could be efficiently implemented – an implementation language should be as efficient as possible to support the general language. The extended part contained low level features to access parts of the hardware that could not be programmed with the general programming language. It was decided to define an implementation language called BETA as the implementation language for GAMMA. The original team consisted of Nygaard, Kristensen and Madsen – Møller-Pedersen joined later in 1976.

As described in Section 3.1 below, the BETA project was based on an initial research idea. This implied that there was much more focus on the BETA project than on the other activities in JLP. For the GAMMA language there were no initial ideas except designing a new language as a successor of SIMULA based on experience with DELTA and some of the best ideas of Pascal. In retrospect, language projects, like most other projects, should be based on one or more good ideas – otherwise they easily end up as nothing more than discussion forums. JLP was a useful forum for discussion of language ideas, but only the BETA project survived.

2.5 The BETA name and language levels

The name BETA was derived from a classification of language levels introduced by Nygaard, introducing a number of levels among existing and new programming languages. The classification by such levels would support the understanding of the nature and purpose of individual languages. The classification also motivated the existence of important language levels.

- The δ -level contains languages for system description and has the DELTA language as an example. A main characteristic of this level is that languages are non-executable.
- The γ -level contains general-purpose programming languages. SIMULA, Algol, Pascal, etc. are all examples of such languages. The JLP project was supposed to develop a new language to be called GAMMA. Languages at this level are by nature executable.
- The β -level contains implementation languages – and BETA was supposed to be a language at this level.
- The α -level contains assembly languages – it is seen as the basic “machine” level at which the actual translation takes place and at which the systems are run.

The level sequence defines the name of the BETA language, although the letter β was replaced by a spelling of the Greek letter β . Other names were proposed and discussed from time to time during the development of BETA. At some point the notion of beta-software became a standard term and this created a lot of confusion and jokes about the BETA language and motivated another name. For many years the name SCALA was a candidate for a new name – SCALA could mean SCAndinavian Language, and in Latin it means ladder and could be interpreted as meaning something ‘going up’. The name of a language is important in order to spread the news appropriately, but names somehow also appear out of the blue and tend to have lives of their own. BETA was furthermore well known at that time and it was decided that it did not make sense to reintroduce BETA under a new name.

3. The BETA project

The original idea for BETA was that it should be an implementation language for a family of application languages at the GAMMA level. Quite early² during the development of BETA, however, it became apparent that there was no reason to consider BETA ‘just’ an implementation language. After the point when BETA was considered a general programming language, we considered it (instead of GAMMA) to be the successor of SIMULA. There were good reasons to consider a successor to SIMULA; SIMULA contains Algol as a subset, and there was a need to simplify parts of SIMULA in much the same way as Pascal is a simplification of Algol. In addition we thought that the new ideas arriving with BETA would justify a new language in the SIMULA style.

3.1 Research approach

The approach to language design used for BETA was naturally highly influenced by the SIMULA tradition. The

² In late 1978 and early 1979.

SIMULA I language report of 1965 opens with these sentences:

“The two main objectives of the SIMULA language are:

- To provide a language for a precise and standardised description of a wide class of phenomena, belonging to what we may call “discrete event systems”.
- To provide a programming language for an easy generation of simulation programs for “discrete event systems”.”

Thus, SIMULA I was considered as a language for system description as well as for programming. It was therefore obvious from the beginning that BETA should be used for system description as well as for programming.

In the '70s the SIMULA/BETA communities used the term system description to correspond to the term model (analysis and design models) used in most methodologies. We have always found it difficult to distinguish analysis, design and implementation. This was because we saw programming as modeling and program executions as models of relevant parts of the application domain. We considered analysis, design and implementation as programming at different abstraction levels.

The original goal for JLP and the GAMMA subcontract was to develop a general purpose programming language as a successor to SIMULA. From the point in time where BETA was no longer just considered to be an implementation language, the research goals for BETA were supplemented by those for GAMMA. All together, the research approach was based on the following assumptions and ideas:

- BETA should be a modeling language.
- BETA should be a programming language. The most important initial idea was to design a language based on one abstraction mechanism. In addition BETA should support concurrent programming based on the coroutine mechanisms of SIMULA.
- BETA should have an efficient implementation.

3.1.1 Modeling and conceptual framework

Creating a model of part of an application domain is always based on certain conceptual means used by the modeler. In this way modeling defines the perspective of the programmer in the programming process. Object-oriented programming is seen as one perspective on programming identifying the underlying model of the language and executions of corresponding programs.

Although it was realized from the beginning of the SIMULA era (including the time when concepts for record handling were developed by Hoare [52-54]) that the class/subclass mechanism was useful for representing concepts including generalizations and specializations, there was no explicit formulation of a conceptual

framework for object-oriented programming. The term object-oriented programming was not in use at that time and neither were terms such as generalization and specialization. SIMULA was a programming language like Algol, Pascal and FORTRAN – it was considered superior in many aspects, but there was no formulation of an object-oriented perspective distinguishing SIMULA from procedural languages.

In the early seventies, the notion of functional programming arrived, motivated by the many problems with software development in traditional procedural languages. One of the strengths of functional programming was that it was based on a sound mathematical foundation (perspective). Later Prolog and other logic programming languages arrived, also based on a mathematical framework.

We did not see functional or logic programming as the solution: the whole idea of eliminating state from the program execution was contrary to our experience of the benefits of objects. We saw functional/logic programming and the development of object-oriented programming as two different attempts to remedy the problems with variables in traditional programming. In functional/logic programming mutable variables are eliminated – in object-oriented programming they are generalized into objects. We return to this issue in Section 4.2.

For object-oriented programming the problem was that there was no underlying sound perspective. It became a goal of the BETA project to formulate such a conceptual framework for object-oriented programming.

The modeling approach to designing a programming language provides overall criteria for the elements of the language. Often a programming language is designed as a layer on top of the computer; this implies that language mechanisms often are designed from technical criteria. BETA was to fulfill both kinds of criteria.

3.1.2 One abstraction mechanism

The original design idea for BETA was to develop a language with only one abstraction mechanism: the pattern. The idea was that patterns should unify abstraction mechanisms such as class, procedure, function, type, and record. Our ambition was to develop the ultimate abstraction mechanism that subsumed all other abstraction mechanisms. In the DELTA report, the term pattern is used as a common term for class, procedure, etc. According to Nygaard the term pattern was also used in the final stages of the SIMULA project. For SIMULA and DELTA there was, however, no attempt to define a language mechanism for pattern.

The reason for using the term pattern was the observation that e.g. class and procedure have some common aspects: they are templates that may be used to create instances. The

instances of a class are objects and the instances of procedures are activation records.

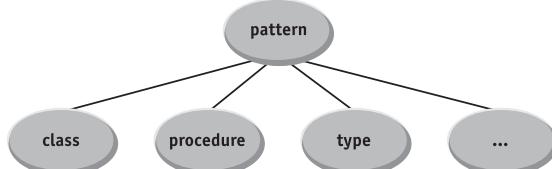


Figure 1 Classification of patterns

In the beginning it was assumed that BETA would provide other abstraction mechanisms as specializations (subpatterns) of the general pattern concept illustrated in Figure 1. In other words, BETA was initially envisaged as containing specialized patterns like class, procedure, type, etc. A subpattern of class as in

```
MyClass: class (# ... #)
```

would then correspond to a class definition in SIMULA. In a similar way a subpattern of procedure would then correspond to a procedure declaration. It should be possible to use a general pattern as a class, procedure, etc. As mentioned in Section 5.8.5, such specialized patterns were never introduced.

Given abstraction mechanisms like class, procedure, function, type and process type, the brute-force approach to unification would be to merge the elements of the syntax for all of these into a syntax describing a pattern. The danger with this approach might be that when a pattern is used e.g. as a class, only some parts of the syntactic elements might be meaningful. In addition, if the unification is no more than the union of class, procedure, etc., then very little has been gained.

The challenges of defining a general pattern mechanism may then be stated as follows:

- The pattern mechanism should be the ultimate abstraction mechanism, subsuming all other known abstraction mechanisms.
- The unification should be more than just the union of existing mechanisms.
- All parts of a pattern should be meaningful, no matter how the pattern is applied.

The design of the pattern mechanism thus implied a heavy focus on abstraction mechanisms, unification, and orthogonality. Orthogonality and unification are closely associated, and sometimes they may be hard to distinguish.

3.1.3 Concurrency

It was from the beginning decided that BETA should be a concurrent programming language. As mentioned, SIMULA supported the notion of quasi-parallel system, which essentially defines a process concept and a cooperative scheduling mechanism. A SIMULA object is a

coroutine and a quasi-parallel process is defined as an abstraction (in the form of a class) on top of coroutines.

The support for implementing hierarchical schedulers was one of the strengths of SIMULA; this was heavily used when writing simulation packages. Full concurrency was added to SIMULA in 1995 by the group at Lund University headed by Boris Magnusson [158].

Conceptually, the SIMULA coroutine mechanism appears simple and elegant, but certain technical details are quite complicated. For BETA, the SIMULA coroutine mechanism was an obvious platform to build upon. The ambition was to simplify the technical details of coroutines and add support for full concurrency including synchronization and communication. In addition it should be possible to write cooperative as well as pre-emptive schedulers.

3.1.4 Efficiency

Although SIMULA was used by many communities in research institutions and private businesses, it had a relatively small user community. However, it was big enough for a yearly conference for SIMULA users to take place.

One of the problems with making SIMULA more widely used was that it was considered very inefficient. This was mainly due to automatic memory management and garbage collection. Computers at that time were quite slow and had very little memory compared to computers of today. The DEC 10 at DAIMI had 128Kbyte of memory. This made efficient memory management quite challenging.

One implication of this was that object orientation was considered to be quite inefficient by nature. It was therefore an important issue for BETA to design a language that could be efficiently implemented. In fact, it was a goal that it should be possible to write BETA programs with a completely static memory layout.

Another requirement was that BETA should be usable for implementing embedded systems. Embedded systems experts found it provoking that Nygaard would engage in developing languages for embedded systems – they did not think he had the qualifications for this. He may not have had much experience in embedded systems, but he surely had something to contribute. This is an example of the controversies that often appeared around Nygaard.

As time has passed, static memory requirements have become less important. However, this issue may become important again, for example in pervasive computing based on small devices.

3.2 Project organization

The process, intention, and organization of the BETA project appeared to be different from those of many projects today. The project existed through an informal

cooperation between Nygaard and the authors. During the project we had obligations as students, professors or consultants. This implied that the time to be used on the project had to be found in between other activities.

As mentioned, Kristensen, Møller-Pedersen and Madsen were students at DAIMI, Århus. In 1974 Kristensen completed his Masters Thesis on error recovery for LR-parsers. He was employed as assistant professor at the University of Ålborg in 1976. Madsen graduated in 1975, having written a Master's thesis on compiler-writing systems, and continued at DAIMI as assistant professor and later as a PhD student. Møller-Pedersen graduated in 1976 with a Master's thesis on the notion of context, with Nygaard as a supervisor. He was then employed by the NCC, Oslo, in 1976 and joined the BETA project at the same time. Nygaard was a visiting professor at DAIMI in 1974-75 – after that he returned to the NCC and continued at DAIMI as a guest lecturer.

Most meetings took place in either Århus or Oslo, and therefore required a lot of traveling. At that time there was a ferry between Århus and Oslo. It sailed during the night and took 16 hours – we remember many pleasant trips on that ferry – and these trips were a great opportunity to discuss language issues without being disturbed. Later when the ferry was closed we had to use other kinds of transportation that were not as enjoyable.

Funding for traveling and meetings was limited. Research funding was often applied for, but with little success. Despite his great contributions to informatics through the development of the SIMULA languages, Nygaard always had difficulties in getting funding in Norway. The Mjølner project described in Section 3.4 is an exception, by providing major funding for BETA development – however, Nygaard was not directly involved in applying for this funding.

The project involved a mixture of heated discussions about the design of the BETA language and a relaxed, inspiring social life. It seemed that for Nygaard there was very little difference between professional work and leisure.

Meetings. The project consisted of a series of more or less regular meetings with the purpose of discussing language constructs and modeling concepts. Meetings were planned in an ad hoc manner. The number of meetings varied over the years and very little was written or prepared in advance.

Meetings officially took place at NCC or at our universities, but our private homes, trams/buses, restaurants, ferries, and taxies were also seen as natural environments in which the work and discussions could continue – in public places people had to listen to loud, hectic discussions about something that must have appeared as complete nonsense to them. But people were tolerant and seemed to accept this weird group.

In October 1977, Madsen and family decided to stay a month in Oslo to complete the project – Madsen's wife, Marianne was on maternity leave – and they stayed with Møller-Pedersen and his family. This was an enjoyable stay, but very little progress was made with respect to completing BETA – in fact we saw very little of Nygaard during that month.

Discussions. A meeting would typically take place without a predefined agenda and without any common view on what should or could be accomplished throughout the meeting. The meetings were a mixture of serious concentrated discussions of ideas, proposals, previous understanding and existing design and general stuff from the life of the participants.

State-of-the-art relevant research topics were rarely subjects for discussion. Nygaard did not consider this important – at least not for the ongoing discussions and elaboration of ideas. Such knowledge could be relevant later, in relation to publication, but was usually not taken seriously. In some sense Nygaard assumed that we would take care of this. Also established understanding, for example, on the background, motivations and the actual detailed contents of languages like Algol, SIMULA or DELTA was not considered important. It appeared to be much better to develop ideas and justify them without historical knowledge or relationships. The freedom was overwhelming and the possibilities were exhausting.

The real strengths of Nygaard were his ability to discuss language issues at a conceptual level and focus on means for describing real-world systems. Most language designers come from a computer science background and their language design is heavily based on what a computer can do: A programming language is designed as a layer on top of the computer making it easier to program. Nygaard's approach was more of a modeling approach and he was basically interested in means for describing systems. This was evident in the design of SIMULA, which was designed as a simulation language and therefore well suited for modeling real systems.

Plans. There was no clear management of the project, and plans and explicit decisions did not really influence the project. We used a lot of time on planning, but most plans were never carried out. Deadlines were typically controlled by the evolution of the project itself and not by a carefully worked out project plan.

Preparation and writing were in most cases the result of an individual initiative and responsibility. Nygaard was active in writing only in the initial phase of the project. Later on Nygaard's research portfolio mainly took the form of huge stacks of related plastic slides, but typically their relation became clear only at Nygaard's presentations. The

progress and revisions of his understanding of the research were simply captured on excellent slides.

At the beginning of the project it was decided that Kristensen and Madsen should do PhDs based on the project. As a consequence of the lack of project organization, it quickly became clear that this would not work.

The original plan for the BETA project was that '*a firm and complete language definition*' should be ready at the end of 1977 [89]. An important deadline was February 1977 – at that time a first draft of a language definition should be available. In 1977 we were far from a complete language definition and Nygaard did not seem in a hurry to start writing a language definition report. However, two working notes were completed in 1976/77. In Section 3.3 and in Section 5.10, we describe the content of these working notes and other publications and actual events in the project.

Social life. Meetings typically lasted whole days including evenings and nights. In connection with meetings the group often met in our private homes and had dinner together, with nice food and wine. The atmosphere was always very enjoyable but demanding, due to an early-morning start with meetings and late-evening end. Dinner conversation was often mixed with debate about current issues of language design. Our families found the experience interesting and inspiring, but also often weird. Nygaard often invited various guests from his network, typically without our knowing and often announced only in passing. Guests included Carl Hewitt, Bruce Moon, Larry Tesler, Jean Vaucher, Stein Krogdahl, Peter Jensen, and many more. They were all inspiring and the visits were learning experiences. In addition there were many enjoyable incidents as when passengers on a bus to the suburb where Madsen lived watched with surprise and a little fear as Nygaard (tall and insistent) and Hewitt (all dressed in red velour and just as insistent) loudly discussed not commonly understandable concepts on the rear platform of the bus.

Nygaard was an excellent wine connoisseur and arranged wine-tasting parties on several occasions. We were “encouraged” to spend our precious travel money on various selected types of wines, and it was beyond doubt worth it. Often other guests were invited and had similar instructions about which wine to bring. At such parties we would be around 10 people in Nygaard’s flat, sitting around their big dinner table and talking about life in general. The wines would be studied in advance in Hugh Johnson’s “World Atlas of Wine” and some additional descriptions would be shared. The process was controlled and conducted by Nygaard at the head of the table.

Crises. The project meetings could be very frustrating since Nygaard rarely delivered as agreed upon at previous

meetings. This often led to very heated discussions. This seemed to be something that we inherited from the SIMULA project. In one incident Kristensen and Madsen arrived in Oslo at the NCC and during the initial discussions became quite upset with Nygaard and decided to leave the project. They took a taxi to the harbor in order to enter the ferry to Århus. Nygaard, however, followed in another taxi and convinced them to join him for a beer in nearby bar – and he succeeded in convincing them to come back with him.

Crises and jokes were essential elements of meetings and social gatherings. Crises were often due to different expectations to the progress of the language development, unexpected people suddenly brought into the project meetings, and problems with planning of the meeting days. Crises were solved, but typically not with the result that the next similar situation would be tackled differently by Nygaard. Serious arguments about status and plans were often solved by a positive view on the situation together with promises for the future. Jokes formed an essential means of taking ‘revenge’ and thereby to overcome crises. Jokes were on Nygaard in order to expose his less appealing habits, as mentioned above, and were often simple and stupid, probably due to our irritation and desperation. Nygaard was an easy target for practical jokes, because he was always very serious about work, which was not something you joked about.³ On one occasion in Nygaard’s office at Department of Informatics at University of Oslo, the telephone calls that Nygaard had to answer seemed to never end, even if we complained strongly about the situation. One time when Nygaard left the office, we taped the telephone receiver to the base by means of some transparent tape. When Nygaard returned, we arranged for a secretary to call him. As usual Nygaard quickly grabbed the telephone receiver, and he got completely furious because the whole telephone device was in his hand. He tried to wrench the telephone receiver off the base unit, but without success. Nygaard blamed us for the lost call (which could be very important as were the approximately 20 calls earlier this morning) and left the office running to the secretary in order to find out who had called. He returned disappointed and angry, but possibly also a bit more understanding of our complaints. At social events he was on the other hand very entertaining and had a large repertoire of jokes – however, practical jokes were not his forte.

3.3 Project events

In this section we mention important events related to the project process as a supplement to the description in the previous sections. Events related to the development of the

³ We never found out whether or not this was a characteristic of Nygaard or of Norwegians in general ☺.

conceptual framework, the language and its implementation are described in the following sections.

Due to the lack of structure in the project organization, it is difficult to point to specific decisions during the project that influenced the design of BETA. The ambition for the project was to strive for the perfect language and it turned out that this was difficult to achieve through a strict working plan. Sometimes the design of a new programming language consists of selecting a set of known language constructs and the necessary glue for binding them together. For BETA the goal was to go beyond that. This implied that no matter what was decided on deadlines, no decisions were made as long as a satisfactory solution had not been found. In some situations we clearly were hit by the well known saying, ‘The best is the enemy of the good’.

The start of the JLP and the start of the BETA project were clearly important events. There was no explicit decision to terminate the JLP – it just terminated.

As mentioned, two working notes were completed in 1976/1977. The first one was by Peter Jensen and Nygaard [66] and was mainly an argument why the NCC should establish cooperation with other partners in order to implement the BETA system programming language on microcomputers.

First language draft. The second working note was the first publication describing the initial ideas of BETA, called *BETA Language Development – Survey Report, 1, November 1976* [89]. A revised version was published in September 1977.

Draft Proposal of BETA. In 1978 a more complete language description was presented in *DRAFT PROPOSAL for Introduction to the BETA Programming Language as of 1st August 1978* [90] and a set of examples [91]. A grammar was included. Here BETA was still mainly considered an implementation language. The following is stated in the report: “*According to the conventional classification of programming languages BETA is meant to be a system programming language. Its intended use is for programming of operating systems, data base systems, communication systems and for implementing new and existing programming languages. ... The reason for not calling it a system programming language is that it is intended to be more general than often associated with system programming languages. By general is here meant that it will contain as few as possible concepts underlying most programming concepts, but powerful enough to build up these. The BETA language will thus be a kernel of concepts upon which more application oriented languages may be implemented and we do not imagine the language as presented here used for anything but implementation of more suitable languages. This will, however, be straightforward to do by use of a compiler-generator. Using this,*

sets of concepts may be defined in terms of BETA and imbedded in a language. Together with the BETA language it is the intention to propose and provide a ‘standard super BETA’.”

As mentioned, BETA developed without any explicit decision into a full-fledged general programming language. In this process it was realized that GAMMA and special-purpose languages could be implemented as class frameworks in BETA. With regard to class frameworks, SIMULA again provided the inspiration. SIMULA provided class Simulation – a class framework for writing simulation programs. Class Simulation was considered a definition of a special-purpose language for simulation – SIMULA actually has special syntax only meaningful when class Simulation is in use. For BETA it provided the inspiration for work on special-purpose languages. The idea was that a special-purpose language could be defined by means of a syntax definition (in BNF), a semantic definition in terms of a class framework, and a syntax-directed transformation from the syntax to a BETA program using the class framework. This is reflected in the 1978 working note.

First complete language definition. In February 1979 – and revised in April 1979 – the report *BETA Language Proposal* [92] was published. It contained the first attempt at a complete language definition. Here BETA was no longer considered just an implementation language: “*BETA is a general block-structured language in the style of Algol, Simula and Pascal. ... Most of the possibilities of Algol-like sequential languages are present*”. BETA was, however, still considered for use in defining application-oriented languages – corresponding to what are often called domain-specific languages today.

The fact that BETA was considered a successor to SIMULA created some problems at the NCC and the University of Oslo. The SIMULA communities considered SIMULA to be THE language, and with good reason. There were no languages at that time with the qualities of SIMULA and as of today, the SIMULA concepts are still in the core of mainstream languages such as C++, Java and C#.

Many people became angry with Nygaard that he seemed willing to give up on SIMULA. He did not look at it that way – he saw his mission as developing new languages and exploring new ideas. However, it did create difficulties in our relationship with the SIMULA community. SIMULA was at that time a commercial product of the NCC. When it became known that Nygaard was working on a successor for SIMULA, the NCC had to send out a message to its customers saying that the NCC had no intentions of stopping the support of SIMULA.

Around 1980 there was in fact an initiative by the NCC to launch BETA as a language project based on the model used for SIMULA. This included planning a call for a standardization meeting, although no such meeting ever took place. The plan was that BETA should be frozen by the end of 1980 and an implementation project should then be started by the NCC. However, none of this did happen.

A survey of the BETA Programming Language. In 1981 the report ‘A Survey of the BETA Programming Language’ [93] formed the basis for the first implementation and the first published paper on BETA two years later [95]. As mentioned in Section 6.1, the first implementation was made in 1983.

Several working papers about defining special-purpose languages were written (e.g. [98]), but no real system was ever implemented. A related subject was that the grammar of BETA should be an integrated part of the language. This led to work on program algebras [96] and metaprogramming [120] that made it possible to manipulate BETA programs as data. Some of the inspiration for this work came during a one-year sabbatical that Madsen spent at The Center for Study of Languages and Information at Stanford University in 1984, working with Terry Winograd, Danny Bobrow and José Meseguer.

POPL paper: Abstraction Mechanisms in the BETA Programming Language. An important milestone for BETA was the acceptance of a paper on BETA for POPL in 1983 [95]. We were absolutely thrilled and convinced that BETA would conquer the world. This did not really happen – we were quite disappointed with the relatively little interest the POPL paper created. At the same conference, Peter Wegner presented a paper called *On the Unification of Data and Program Abstractions in Ada* [159]. Wegner’s main message was that Ada contained a proliferation of abstraction mechanisms and there was no uniform treatment of abstraction mechanisms in Ada. Naturally we found Wegner’s paper to be quite in line with the intentions of BETA and this was the start of a long cooperation with Peter Wegner, who helped in promoting BETA.

Hawthorne Workshop. Peter Wegner and Bruce Shriver (who happened to be a visiting professor at DAIMI at the same time as Nygaard) invited us to the Hawthorne workshop on object-oriented programming in 1986. This was one of the first occasions where researchers in OOP had the opportunity to meet and it was quite useful for us. It resulted in the book on Research Directions in Object-Oriented Programming [145] with two papers on BETA [100, 111]. Peter Wegner and Bruce Shriver invited us to publish papers on BETA at the Hawaii International Conference on System Sciences in 1988.

Sequential parts stable. In late 1986/early 1987 the sequential parts of the language were stable, and only minor changes have been made since then.

Multisequential parts stable. A final version of the multisequential parts (coroutines and concurrency) was made in late 1990, early 1991.

BETA Book. Peter Wegner also urged us to write a book on BETA and he was the editor of the BETA book published by Addison Wesley/ACM Press in 1993 [119].

For a number of years we gave BETA tutorials at OOPSLA, starting with OOPSLA’89 in New Orleans. Dave Thomas and others were quite helpful in getting this arranged – especially at OOPSLA’90/ECOOP’90 in Ottawa, he provided excellent support.

At OOPSLA’89 we met with Dave Unger and the Self group; although Self [156] is a prototype-based language and BETA is a class-based language, we have benefited from cooperation with the Self group since then. We believe that Self and BETA are both examples of languages that attempt to be based on simple ideas and principles.

The Mjølner (Section 3.4) project (1986-1991) and the founding of Mjølner Informatics Ltd. (1988) were clearly important for the development of BETA.

Apple and Apollo contracts. During the Mjølner project we got a contract with Apple Computer Europe, Paris, to implement BETA for the Macintosh – Larry Taylor was very helpful in getting this contract. A similar contract was made with Apollo Computer, coordinated by Søren Bry.

In 1994, BETA was selected to be taught at the University of Dortmund. Wilfried Ruplin was the key person in making this happen. A German introduction to programming using BETA was written by Ernst-Erich Doberkat and Stefan Dißmann [38]. This was of great use for the further promotion of BETA as a teaching language.

Dahl & Nygaard receive ACM Turing Award. In 2001 Dahl and Nygaard received the ACM Turing Award (“for their role in the invention of object-oriented programming, the most widely used programming model today”).

Dahl & Nygaard receive the IEEE von Neumann Medal. In 2002 they received the IEEE John von Neumann Medal (“for the introduction of the concepts underlying object-oriented programming through the design and implementation of SIMULA 67”). Dahl was seriously ill at that time so he was not able to attend formal presentations of these awards, including giving the usual Turing Award lecture. Dahl died on June 29, 2002. Nygaard was supposed to give his Turing Award lecture at OOPSLA 2002 in Vancouver, October 2002, but unfortunately he died on August 10, just a few weeks after Dahl. Life is full of strange coincidences. Madsen was invited to give a lecture

at OOPSLA 2002 instead of Nygaard. The overall theme for that talk was ‘*To program is to understand*’, which in many ways summarizes Nygaard’s approach to programming. One of Nygaard’s latest public appearances was his after-dinner talk at ECOOP 2002 in Malaga, where he gave one of his usual entertaining talks that even the spouses enjoyed.

3.4 The Mjølner Project

Mjølner Project start. The Mjølner⁴ Project (1986-1991) [76] was an important step in the development of BETA. The objective of the Mjølner project was to increase the productivity of high-quality software in industrial settings by designing and implementing object-oriented software development environments supporting specification, implementation and maintenance of large production programs. The project was carried out in cooperation between Nordic universities and industrial companies with participants from Denmark, Sweden, Norway and Finland. In the project three software development environments were developed:

- **Object-oriented SDL and tools:** The development of Object-oriented SDL is described in Section 7.3.
- **The Mjølner Orm System:** a grammar-based interactive, integrated, incremental environment for object-oriented languages. The main use of Orm was to develop an environment for SIMULA.
- **The Mjølner BETA System:** a programming environment for BETA.

Mjølner Book. The approach to programming environments developed within the Mjølner Project is documented in the book *Object-Oriented Environments – the Mjølner Approach* [76], covering all these three developments.

The development of the Mjølner BETA System was in a Scandinavian context a large project. The project was a major reason for the success of BETA. During this project the language developed in the sense that many details were clarified. For example, the Ada-like rendezvous for communication and synchronization was abandoned in favor of semaphores, pattern variables were introduced, etc. It was also during the Mjølner project that the fragment system found its current form – cf. Section 5.8.4.

Most of the implementation techniques for BETA were developed during the Mjølner project, together with native compilers for Sun, Macintosh, etc. Section 6 contains a description of the implementation.

⁴ In the Nordic myths Mjølner is the name of Thor’s hammer; Thor is the Nordic god of thunder. Mjølner is the perfect tool: it grows with the task, always hits the target, and always returns safely to Thor’s hand.

A complete programming environment for BETA was developed. In addition to compilers there was a large collection of libraries and application frameworks including a meta-programming system called Yggdrasil⁵, a persistent object store with an object browser, and application frameworks for GUI programs built on top of Athena, Motif, Macintosh and Windows. A platform independent GUI framework with the look and feel of the actual platform was developed for Macintosh, Windows and UNIX/Motif.

The environment also included the MjølnerTool, which was an integration of the following tools: a source code browser called Ymer, an integrated text- and syntax-directed editor called Sif, a debugger called Valhalla, an interface builder called Frigg, and a CASE tool called Freja supporting a graphical syntax for BETA – see also Section 4.5.

Mjølner Informatics. The Mjølner BETA System led in 1988 to the founding of the company Mjølner Informatics Ltd., which for many years developed and marketed the Mjølner BETA System as a commercial product. Sales of the system never generated a high profit, but it gave Mjølner Informatics a good image as a business, and this attracted a lot of other customers. Today the Mjølner BETA System is no longer a commercial product, but free versions may be obtained from DAIMI.

It may seem strange from the outside that three environments were developed in the Mjølner Project. And it is indeed strange. SDL was, however, heavily used by the telecommunication industry, and there was no way to replace it by say BETA – the only way to introduce object orientation in that industry seemed to be by adding object orientation to SDL. Although SDL had a graphical syntax, it also had a textual syntax, and it had a well-defined execution semantics, so it was more or less a domain-specific programming language (the domain being telecommunications) and not a modeling language. Code generators were available for different (and at that time specialized) platforms. SDL is still used when code generation is needed, but for modeling purposes UML has taken over. UML2.0 includes most of the modeling mechanisms of SDL, but not the execution semantics.

The BETA team did propose to the people in charge of the Mjølner Orm development – which focused on SIMULA – that they join the BETA team, and that we concentrate on developing an environment for BETA. BETA was designed as a successor of SIMULA, and we found that it would be better to just focus on BETA. However, the SIMULA people were not convinced; it is often said that SIMULA, like Algol 60, is one of the few languages that is better than

⁵ Most tools in the Mjølner System had names from Nordic mythology.

most of its successors – we are not the ones to judge about this with respect to BETA. The lesson here is perhaps that in a project like Mjølner more long-term goals would have been beneficial. If the project had decided to develop one language including a graphical notation that could replace SDL, then this language might have had a better chance to influence the industry than each of OSDL, SIMULA and BETA.

The motivation for modeling languages like SDL (and later UML) was that industries wanted to be independent of (changing) programming languages and run-time environments. A single language like BETA that claims to be both a programming language and a modeling language was therefore not understood. Even Java has not managed to get such a position. It is also interesting to note that while the Object Management Group advocates a single modeling language, covering many programming languages and platforms, Microsoft advocates a single programming language (or rather a common language run-time, CLR) on top of which they want to put whatever domain-specific modeling language the users in a specific domain require.

4. Modeling and conceptual framework

We believe that the success of object-oriented programming can be traced back to its roots in simulation. SIMULA I was designed to describe (model) real-world systems and simulate these systems on a computer. This eventually led to the design of SIMULA 67 as a general programming language. Objects and classes are well suited for representing phenomena and concepts from the real world and for programming in general. Smalltalk further refined the object model and Alan Kay described object-oriented programming as a view on *computation as simulation* [68] (see also the formulation by Tim Budd [22]). An important aspect of program development is to understand, describe and communicate about the application domain and BETA should be well suited for this. In the BETA book [119] (page 3) this is said in the following way:

To program is to understand: The development of an information system is not just a matter of writing a program that does the job. It is of utmost importance that development of this program has revealed an in-depth understanding of the application domain; otherwise, the information system will probably not fit into the organization. During the development of such systems it is important that descriptions of the application domain are communicated between system specialists and the organization.

The term “*To program is to understand*” has been a leading guideline for the BETA project. This implied that an essential part of the BETA project was the development of a conceptual framework for understanding and organizing

knowledge about the real world. The conceptual framework should define the object-oriented perspective on programming and provide a semantic foundation for BETA. Over the years perhaps more time was spent on discussing the conceptual framework than the actual language. Issues of this kind are highly philosophical and, not being philosophers, we could spend a large amount of time on this without progress.

Since BETA was intended for modeling as well as programming there was a rule that applied when discussing candidates for language constructs in BETA: a given language construct should be motivated from both the modeling and the programming point of view. We realized that many programmers did not care about modeling but were only interested in technical aspects of a given language – i.e. what you can actually express in the language. We thus determined that BETA should be usable as a ‘normal’ programming language without its capabilities as a modeling language. We were often in the situation that something seemed useful from a modeling point of view, but did not benefit the programming part of the language and vice versa.

We find the conceptual framework for BETA just as important as the language – in this paper we will not go into details, but instead refer to chapters 2 and 18 in the book on BETA [119]. Below we will describe part of the rationale and elements of the history of the conceptual framework. In Section 5, where the rationale for the BETA language is described, we will attempt to describe how the emphasis on modeling influenced the language.

4.1 Programming as modeling

As mentioned, DELTA was one of the starting points for the BETA project. For a detailed description of DELTA the reader is referred to the DELTA report [60]. Here we briefly summarize the concepts that turned out to be most important for BETA.

The system to be described was called the *referent system*. A referent system exists in what today’s methodologies call application domain. A description of a system – the *system description* – is a text, a set of diagrams or a combination describing the aspects of the system to be considered. Given a system description, a *system generator* may generate a *model system* that simulates the considered aspects of the referent system. These concepts were derived from the experience of people writing simulation programs (system descriptions) in SIMULA and running these simulations (model systems).

Programming was considered a special case of system description – a program was considered a system description and a program execution was considered a model system. Figure 2 illustrates the relationship between the referent system and the model system.

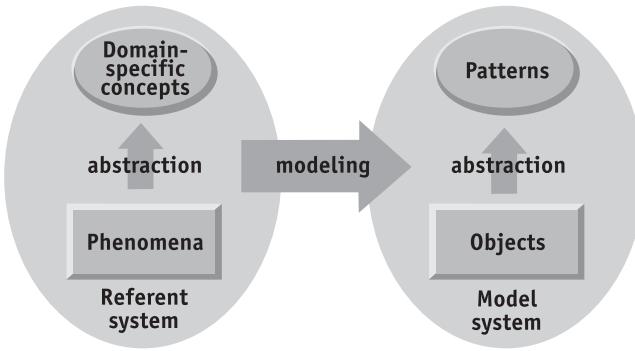


Figure 2 Modeling

As illustrated in Figure 2, phenomena and (domain-specific) concepts from the referent system are identified and represented as (realized) objects and concepts (in the form of patterns) in the model system (the program execution). The modeling activity of Figure 2 includes the making of a system description and having a system generator generate the model system according to this description.

4.2 Object-orientation as physical modeling

For BETA it has been essential to make a clear distinction between the program and the program execution (the model system). A program is a description in the form of a text, diagrams or a combination – the program execution is the dynamic process generated by the computer when executing the program. At the time when BETA was developed, many researchers in programming and programming languages were focusing on the program text. They worked on the assumption that properties of a program execution could (and should) be derived from analysis of the program text, including the use of assertions and invariants, formal proofs and formal semantics. Focusing on the (static) program text often made it difficult to explain the dynamics of a program execution. Especially for object-oriented programming, grasping the dynamic structure of objects is helped by considering the program execution. But considering the program execution is also important in order to understand mechanisms such as recursion and block structure.

The discussion of possible elements in the dynamic structure of a BETA program execution was central during the design of BETA. This included the structure of coroutines (as described in Section 5.7 below), stacks of activation records, nested objects, references between objects, etc. Many people often felt that we were discussing implementation, but for us it was the semantics of BETA. It did cover aspects that normally belonged to implementation, but the general approach was to identify elements of the program execution that could explain to the programmer how a BETA program was executing.

At that time, formal semantics of programming languages was an important issue and we were often confronted with the statement that we should concentrate on defining a formal semantics for BETA. Our answer to that was vague in the sense that we were perhaps uncertain whether or not they were right, but on the other hand we had no idea how to approach a formal semantics for a language we were currently designing. It seemed to us that the current semantic models just covered well known language constructs and we were attempting to identify new constructs. Also, our mathematical abilities were perhaps not adequate to mastering the mathematical models used at that time for defining formal semantics.

Many years later we realized that our approach to identifying elements of the program execution might be seen as an attempt to define the semantics of BETA – not in a formal way, but in a precise and conceptual way.

The focus on the program execution as a model eventually led to a definition of object-oriented programming based on the notion of physical model – first published at ECOOP in '88 [116]:

Object-oriented programming. A program execution is regarded as a physical model simulating the behavior of either a real or imaginary part of the world.

The notion of physical is essential here. We considered (and still do) objects as physical material used to construct models of the relevant part of the application domain. The analogy is the use of physical material to construct models made of cardboard, wood, plastic, wire, plaster, LEGO bricks or other substances. Work on object-oriented programming and computerized shared material by Pål Sørgaard [149] was an essential contribution here.

Webster defines a model in the following way: “In general a model refers to a small, abstract or actual representation of a planned or existing entity or system from a particular viewpoint” [1]. Mathematical models are examples of abstract representations whereas models of buildings and bridges made of physical material such as wood, plastic, and cartoon are examples of actual representations. Models may be made of existing (real) systems as in physics, chemistry and biology, or of planned (imaginary) systems like buildings, and bridges.

We consider object-oriented models⁶ to be actual (physical) representations made from objects. An object-oriented model may be of an existing or planned system, or a

⁶ The term modeling is perhaps somehow misleading since the model eventually becomes the real thing – in contrast to models in science, engineering and architecture. We originally used the term *description*, in SIMULA and DELTA terminology, but changed to modeling when OOA/OOD and UML became popular.

combination. It may be a reimplementation of a manual system on a computer. An example may be a manual library system that is transferred to computers. In most cases, however, a new (planned) system is developed. In any case, the objects and patterns of the system (model) represent phenomena and concepts from the application domain. An object-oriented model furthermore has the property that it may be executed and simulate the behavior of the system in accordance with the computation-is-simulation view mentioned above.

The application domain relates to the real world in various ways. Most people would agree that a library system deals with real world concepts and phenomena such as books and loans. Even more technical domains like a system controlling audio/video units and media servers deal with real-world concepts and phenomena. Some people might find that a network communication protocol implementing TCP/IP is not part of the real world, but it definitely becomes the real world for network professionals, just as an electronic patient record is the real world for healthcare professionals. Put in other words: Even though the real world contains real trees and not so many binary search trees or other kinds of data structures, the modeling approach is just as valuable for such classical elements of (in this case) the implementation domain.

In any case the modeling approach should be the same for all kinds of application domains – this is also the case for the conceptual means used to understand and organize knowledge about the application domain, be it the real world or a technical domain. In the approach taken by OO and BETA we apply conceptual means used for organizing knowledge about the real world, as we think this is useful for more technical and implementation-oriented domains as well. In Chapter 5 we describe how the modeling approach has influenced the design of the BETA language.

From the above definition it should be evident that phenomena that have the property of being physical material should be represented as objects. There are, however, other kinds of phenomena in the real world. This led to a characterization of the essential qualities of phenomena in the real world systems of interest for object-oriented models:

- Substance – the physical material transformed by the process.
- Measurable properties of the substance.
- Transformations of the substance.

People, vehicles, and medical records are examples of phenomena with substance, and they may be represented by objects in a program execution. The age, weight or blood pressure of a person are examples of measurable properties of a person and may be represented by values (defined by value types) and/or functions. Transformations of the

substance may be represented by the concurrent processes and procedures being executed as part of the program execution. The understanding of the above qualities had a profound influence on the semantics of BETA.

4.3 Relation to other perspectives

In order to arrive at a conceptual understanding of object orientation, we found it important to understand the differences between object-orientation and other perspectives such as procedural, functional, and constraint programming. We thus contrasted our definition of object orientation (see e.g. our ECOOP'88 paper [116] and Chapter 2 in the BETA book [119]) to similar definitions for other perspectives. In our understanding, the essential differences between procedural and functional programming related to the use of mutable variables. In procedural programming a program manipulates a set of mutable variables. In pure functional programming there is no notion of mutable variable. A function computes its result solely based on its arguments. This also makes it easy to formulate a sound mathematical foundation for functional programming. We are aware that our conception of functional programming may not correspond to other people's understanding. In most functional programming languages you may have mutable variables and by means of closures you may even define object-oriented programming language constructs as in CommonLisp. However, if you make use of mutable variables it is hard to distinguish functional programming from procedural programming. Another common characteristic of functional languages is the strong support for higher functions and types. However, higher-order functions (and procedures) and types may be used in procedural as well as object-oriented programming. Algol and Pascal support a limited form of higher-order functions and procedures, and generic types are known from several procedural languages. Eiffel and BETA are examples of languages supporting generic classes (corresponding to higher-order types), and for BETA it was a goal to support higher-order functions and procedures. When we discuss functional programming in this paper, it should be understood in its pure form where a function computes its result solely based on its arguments. This includes languages using non-mutable variables as in `let x=e1 in e2`.

For BETA it was not a goal to define a pure object-oriented language as it may have been for Smalltalk. On the contrary, we were interested in integrating the best from all perspectives into BETA. We thus worked on developing an understanding of a unified approach that integrated object-oriented programming with functional, logic and procedural programming [116]. BETA supports procedural programming and to some extent functional programming. We also had discussions with Alan Borning and Bjorn Freeman-Benson on integrating constraint-oriented programming into BETA. The idea of using equations

(constraints) to describe the state of objects was very appealing, but we never managed to identify primitive⁷ language constructs that could support constraints. However, a number of frameworks supporting constraints were developed by students in Aarhus.

4.4 Concepts and abstraction

It was of course evident from the beginning that the class/subclass constructs of SIMULA were well suited to representing traditional Aristotelian concepts (for a description of Aristotelian concepts, see the BETA book) including hierarchical concepts. The first example of a subclass hierarchy was a classification of vehicles as shown in Figure 3.

Abstraction is perhaps the most powerful tool available to the human intellect for understanding complex phenomena. An abstraction corresponds to a concept. In the Scandinavian object-oriented community it was realized in the late seventies by a number of people, including the authors and collaborators, that in order to be able to create models of parts of the real world, it was necessary to develop an explicit understanding of how concepts and phenomena relate to object-oriented programming.

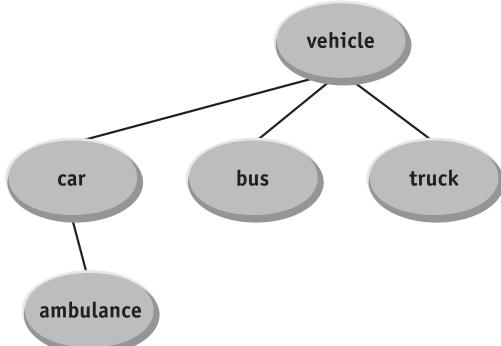


Figure 3 Example of a subclass hierarchy

In the late seventies and early eighties the contours of an explicit conceptual framework started to emerge – there was an increasing need to be explicit about the conceptual basis of BETA and object orientation in general. The ongoing discussions on issues such as multiple inheritance clearly meant that there was a need for making the conceptual framework explicit. These discussions eventually led to an explicit formulation of a conceptual framework by means of Aristotelian concepts in terms of intension, extension and designation to be used in object-oriented modeling. An important milestone in this work was the Master's thesis of Jørgen Lindskov Knudsen [71] and part of the PhD Thesis of Jørgen Lindskov Knudsen and Kristine Thomsen [78].

⁷ We do not consider equations to be programming-language primitives.

Knudsen supplemented the conceptual framework with the so-called *prototypical concepts* inspired by Danish philosopher Sten Folke Larsen [42], who argued that most everyday concepts are not Aristotelian but fuzzy (prototypical). An Aristotelian concept is characterized by a set of defining properties (the intension) that are possessed by all phenomena covered by the concept (the extension). For a prototypical concept the intension consists of examples of properties that the phenomena may have, together with a collection of typical phenomena covered by the concept, called *prototypes*. An Aristotelian concept structure has well defined boundaries between the extensions of the concepts, whereas this is not the case for a prototypical concepts structure. In the latter the boundaries are blurred/fuzzy. A class is well suited to representing Aristotelian concepts, but since most everyday concepts are prototypical, a methodology should allow for prototypical concepts to be used during analysis. Prototypical concepts should not be confused with prototype-based languages. A prototypical concept is still a concept – prototypical objects are not based on any notion of concept. Prototypical concepts are described in the BETA book [119], and the relationship between prototypical concepts and prototype-based languages is discussed by Madsen [113].

In the seventies there was similar work on modeling going on in the database and AI communities and some of this work influenced on the BETA project. This included papers such as the one by Smith & Smith [147] on database abstraction.

The realization that everyday concepts were rarely Aristotelian made it clear that it was necessary to develop a conceptual framework that was richer than the current programming language in use. In the early days, there might have been a tendency to believe that SIMULA and other object-oriented languages had all mechanisms that were needed to model the real world – this was of course naive, since all languages put limitations on the aspects of the real world that can be naturally modeled. Programmers have a tendency to develop an understanding of the application domain in terms of elements of their favorite programming language. A Pascal programmer models the real world in terms of Pascal concepts like records and procedures. We believed that the SIMULA concepts (including class, and subclass) were superior to other programming languages with respect to modeling.

The conceptual framework associated with BETA is deliberately developed to be richer than the language. In addition to introducing prototypical concepts, the BETA book discusses different types of classification structures that may be applied to a given domain, including some that cannot be directly represented in mainstream programming languages. The rationale for the richer conceptual framework is that programmers should understand the

application domain by developing concepts without being constrained by the programming language. During implementation it may of course be necessary to map certain concepts into the programming language. It is, however, important to be explicit about this. This aspect was emphasized in a paper on teaching object-oriented programming [77].

4.5 Graphical syntax for modeling

When object-oriented programming started to become mainstream in the early eighties, code reuse by means of inheritance was often seen as the primary advantage of object-oriented programming. The modeling capabilities were rarely mentioned. The interest in using object-oriented concepts for analysis and design that started in the mid-eighties was a positive change since the modeling capabilities came more in focus.

One of the disadvantages of OOA/OOD was that many people apparently associated analysis and design with the use of graphical languages. There is no doubt that diagrams with boxes and arrows are useful when designing systems. In the SIMULA and BETA community, diagrams had of course also been used heavily, but when a design/model becomes stable, a textual representation in the form of an abstract program is often a more compact and comprehensive representation.

The mainstream modeling methodologies all proposed graphical languages for OOA/OOD, which led to the UML effort on designing a standardized graphical language for OOA/OOD. We felt that this was a major step backwards – one of the advantages of object-orientation is that the same languages and concepts can be applied in all phases of the development process, from analysis through design to implementation. By introducing a new graphical language, one reintroduced the problem of different representations of the model and the code. It seems to be common sense that most software development is incremental and iterative, which means that the developer will iterate over analysis, design and implementation several times. It is also generally accepted that design will change during implementation. With different representations of the model and the code it is time consuming to keep both diagrams and code in a consistent state.

In the early phases of Mjølner Project it was decided to introduce a graphical syntax for the abstraction mechanisms of BETA as an alternative to the textual syntax. The Freja CASE tool [141, 142] was developed using this syntax. In addition, Freja was integrated with the text and structure editor in such a way that the programmer could easily alternate between a textual and graphical representation of the code.

When UML became accepted as a common standard notation, the developers at Mjølner Informatics decided to

replace the graphical syntax defined for BETA by a subset of UML. Although major parts of BETA had a one-to-one correspondence with this UML subset, some of the problems of different representations were reintroduced.

It is often said that a picture says more than a thousand words. This is true. Nygaard in his presentations often used a transparency with this statement (and a picture of Madonna). This was always followed by one saying that a word often says more than a thousand pictures, illustrated by a number of drawings of vehicles and the word ‘vehicle’. The point is that we use words to capture essential concepts and phenomena – as soon as we have identified a concept and found a word for it, this word is an efficient means for communication among people. The same is true in software design. In the initial phase it is useful to use diagrams to illustrate the design. When the design stabilizes it is often more efficient to use a textual representation for communication between the developers. The graphical representation may still be useful when introducing new people to the design.

4.6 Additional notes

It was often difficult to convey to other researchers what we understood by system description and why we considered it important. As mentioned, there was an important workshop at the IBM Hawthorne Research Center in New York in 1986, organized by Peter Wegner and Bruce Shriver, in which Dahl, Nygaard and Madsen participated. Here we had long and heated debates with many researchers – it was difficult to agree on many issues, most notably the concept of multiple inheritance. We later realized that for most people at that time the advantage of object-orientation was from a reuse point of view – a purely technical argument. For us, coming from the SIMULA tradition, the modeling aspect was at least as important, but the difference in perspective was not explicit. Later Steve Cook [26] made the difference explicit by introducing the ideas of the ‘Scandinavian School’ and the ‘U.S. School’ of object-orientation.

At that time the dominant methodology was based on structured analysis and design followed by implementation – SA/SD [162]. SIMULA users rarely used SA/SD, but formulated their designs directly in SIMULA. The work on DELTA and system description was an attempt to formulate concepts and languages for analysis and design – Peter Wegner later said that SIMULA was a language with a built-in methodology. We did find the method developed by Michael Jackson [63] more interesting than SA/SD. In SA/SD there is focus on identifying functionality. In Jackson’s method a model of the application domain is first constructed and functionality is then added to this model. The focus on modeling was in much more agreement with our understanding of object-orientation.

In the mid-eighties, Yourdon and others converted to object orientation and published books on object-oriented analysis and design, e.g. [25]. This was in many ways a good turning point for object orientation, because many more people now started to understand and appreciate its modeling advantages.

In 1989 Madsen was asked to give a three-day course on OOD for software developers from Danish industry. He designed a series of lectures based on the abstraction mechanisms of BETA – including the conceptual framework. At the end of the first day, most of the attendees complained that this was not a design course, but a programming course. The attendees were used to SA/SD and had difficulties in accepting the smooth transition from design to implementation in object-oriented languages – it should be said that Madsen was not trying to be very explicit about this. There was no tradition for this in the SIMULA/BETA community – design was programming at a higher level of abstraction.

It actually helped that, after some heated discussions with some of the attendees, a person stood up in the back of the room presenting himself and two others as being from DSB (the Danish railroad company) – he said that his group was using SIMULA for software development and they have been doing design for more than 10 years in the way it had been presented. He said that it was very difficult for them to survive in a world of SA/SD where SIMULA was quite like a stepchild – the only available SIMULA compiler was for a DEC 10/20 which was no longer in production, and they therefore had to use the clone produced by a third party. However, together with the course organizer, Andreas Munk Madsen, Madsen redesigned the next two days' presentations overnight to make more explicit why this was a course on design.

The huge interest in modeling based on object orientation in the late eighties was of course positive. The disadvantage was that now everybody seemed to advocate object orientation just because it had become mainstream. There were supporters (or followers) of object-orientation who started to claim that the world *is* object-oriented. This is of course wrong – object orientation is a *perspective* that one may use when modeling the world. There are many other perspectives that may be used to understand phenomena and concepts of the real world.

5. The Language

In this section we describe the rationale for the most important parts of BETA. We have attempted to make this section readable without a prior knowledge of BETA, although some knowledge of BETA will be an advantage. The reader may consult the BETA book [119] for an introduction to BETA.

The BETA language has evolved over many years and many changes to the semantics and syntax have appeared in this period. It would be too comprehensive to describe all of the major versions of BETA in detail. We will thus describe BETA as of today, with emphasis on the rationale and discussions leading to the current design and to intermediate designs. In Section 5.10, we will briefly describe the various stages in the history of the language.

As mentioned in Section 3.1, most language mechanisms in BETA are justified from a technical as well as a modeling point of view. In the following we will attempt to state the technical as well as the modeling arguments for the language mechanisms being presented.

5.1 One abstraction mechanism

From the beginning the challenge was to design a programming language mechanism called a *pattern* that would subsume well-known abstraction mechanisms. The common characteristic of abstraction mechanisms is that they are *templates* for generating *instances* of some kind. In the mid seventies when the BETA project started, designers and programmers were not always explicit about whether or not a given construct defined a template or an instance and when a given instance was generated. In this section we describe the background, rationale and final design of the pattern.

5.1.1 Examples of abstraction mechanisms

When the BETA project was started, research in programming languages was concerned with a number of abstraction mechanisms. Below we describe some of the abstraction mechanisms that were discussed in the beginning of the BETA project. We will explicitly use a terminology that distinguishes templates from instances.

Record type. A record type as known from Pascal defines a list of fields of possibly different types. The following is an example of a Pascal record type, which is a template for records:

```
type Person =
  record name: String; age: integer end;
```

Instances of Person may be defined as follows:

```
var P:Person;
```

Fields of the record P may be read or assigned as follows:

```
n := P.name; P.age := 16
```

Value type. Value types representing numbers, Boolean values, etc. have always been important in programming languages. New abstraction mechanisms for other kinds of value types were proposed by many people. This included compound value types like complex number, enumeration types such as color known from Pascal and numbers with a unit such as speed. The main characteristic of a value type is that it defines a set of values that are assignable and

comparable. Value types may to some extent be defined by means of records and classes, but as mentioned in Section 2.4, we did not think that this was a satisfactory solution. We return to this in Section 5.8.2.

Procedure/function. A procedure/function may be viewed as a template for activation records. It is defined by a name, input arguments, a possible return type, and a sequence of statements that can be executed. A typical procedure in a Pascal-like language may look like

```
integer distance(var p1,p2: Point)
  var dist: real
begin ...; return dist; end
```

A procedure call of the form `d := distance(x,y)` generates an instance in the form of an activation record for `distance`, transmits `x` and `y` to the activation record, executes the statement part and returns a value to be assigned to `d`.

The notion of pure function (cf. Section 4.2) was also considered an abstraction mechanism that should be covered by the pattern.

Class. A (simple) class in the style of SIMULA has a name, input arguments, a possible superclass, a set of data fields, and a set of operations. Operations are procedures or functions in the Algol style. In today's object-orientation terminology the operations are called methods. One of the uses of class was as a mechanism for defining abstract data types.

Module. The module concept was among others proposed as an alternative to the class as a means for defining abstract data types. One of the problems with module – as we saw it – was that it was often not explicit from the language definition whether a module was a template or an instance. As described in Section 5.8.8, we considered a module to be an instance rather than a template.

Control abstraction. A control abstraction defines a control structure. Over the years a large variety of control structures have been proposed. For BETA it was a goal to be able to define control abstractions. Control abstractions were mainly found in languages like CLU that allowed iterators to be defined on sets of objects.

Process type. A process type defines a template for either a coroutine or a concurrent process. In some languages, however, a process declaration defined an instance and not a template. In SIMULA, any object is in fact a coroutine and a SIMULA class defines a sequence of statements much like a procedure. A SIMULA class may in this sense be considered a (pseudo) process type. For a description of the SIMULA coroutine mechanism see e.g. Dahl and Hoare [30]. In Concurrent Pascal the SIMULA class concept was generalized into a true concurrent process type [17].

The relationship between template and instance for the above abstraction mechanisms is summarized in the table below:

Abstraction/template	Instance
record type	record
value type	value
procedure/function	activation record
class	object
control abstraction	control activation
module?	module
process type	process object

The following observations may further explain the view on abstraction mechanisms and instances in the early part of the BETA project:

- Some of terms in the above table were rarely considered by others at the programming level, but were considered implementation details. This is the case for activation record, control activation and process object. As we discuss elsewhere, we put much focus on the program execution – the dynamic evolution of objects and actions being executed – for understanding the meaning of a program. This was in contrast to most programming-language schools where the focus was on the program text.
- The notion of value type might seem trivial and just a special case of record type. However, as mentioned in Section 2.4, Nygaard found it doubtful to use the class concept to define value types – we return to this subject in Section 5.8.2.
- A record type is obviously a special case of a class in the sense that a class may ‘just’ define a list of data fields. The only reason to mention record type as a case here is that the borderline between record type, value type and class was not clear to us.
- If one follows Hoare, an object or abstract data type could only be accessed via its operations. We found it very heavyweight to insist that classes defining simple record types should also define accessor functions for its fields. This issue is further discussed in Section 5.5.

5.1.2 Expected benefits from the unification

As mentioned previously, the pattern should be more than just the union of the above abstraction mechanisms. Below we list some language features and associated issues that should be considered.

- **Pattern.** The immediate benefit of unifying class, procedure, etc. is that this ensures a uniform treatment of all abstraction mechanisms. At the conceptual level,

programmers have a general concept covering all abstraction mechanisms. This emphasizes the similarities among class, procedure, etc., with respect to being abstractions defining templates for instances. From a technical point of view, it ensures orthogonality among class, procedure, etc.

- **Subpattern.** It should be possible to define a pattern as a subpattern of another pattern. This is needed to support the notion of subclass. From the point of view of orthogonality, this means that the notion of subpattern must also be meaningful for the other abstraction mechanisms. For example, since a procedure is a kind of pattern, inheritance for procedures must be defined – and in a way that makes it useful.
- **Virtual pattern.** To support virtual procedures, it must be possible to specify virtual patterns. Again, the concept of virtual pattern must be meaningful for the other abstraction mechanisms as well. As a virtual pattern can be used as a class, the concept of virtual class must be given a useful meaning. It turned out that the notion of virtual class (or virtual type) was perhaps one of the most useful contributions of BETA.
- **Nested pattern.** Since Algol, SIMULA, and DELTA are block-structured languages that support nesting of procedures and classes, it was obvious that BETA should also be a block-structured language. I.e., it should be possible to nest patterns arbitrarily.

- **Pattern variable.** Languages like C contain pointers to procedures. For BETA, procedure typed variables were not considered initially, but later suggested by Ole Agesen, Svend Frølund and Michael H. Olsen in their Master's thesis on persistent objects [4]. The uniformity of BETA implied that we then had classes, procedures, etc. as first-order values.

In addition to being able to unify the various abstraction mechanisms, it was also a goal to be able to describe objects directly without having to define a pattern and generate an instance. This lead to the notion of singular objects:

- **Singular objects.** In Algol and SIMULA it is possible to have inner blocks. In Pascal it is possible to define a record variable without defining a record type. For BETA it was a design goal that singular objects (called *anonymous classes* in Java and Scala [133-135]) should apply for all uses of a pattern. That is, it should be possible to write a complete BETA program in the form of singular objects – without defining any patterns at all.

5.1.3 Similarities between object and activation record

As mentioned in Section 3.1.2, the observation about the similarities between objects and activation records was one of the main motivations for unifying e.g. class and

procedure. From the beginning the following similarities between objects and activation records were observed:

- An *object* is generated as an instance of a *class*. An *activation record* is generated as part of a *procedure invocation*. In both cases *input parameters* may be transferred to the object/activation record.
- An object consists of *parameters*, and *data items* (fields). An activation record also consists of *parameters* and *data items* in the form of local variables.
- An object may contain *local procedures* (methods). In a block-structured language an activation record may have *local (nested) procedures*.
- In a block-structured language, an activation record may have a pointer to the statically enclosing activation record (often called the *static link* or *origin*). In SIMULA, classes may be nested, so a SIMULA object may also have an origin.
- An activation record may have a reference pointing to the activation record of the calling procedure (often called the *dynamic link* or *caller*). In most object-oriented languages there is no counterpart to a dynamic link in an object. In SIMULA this is different since a SIMULA object is potentially a coroutine.

Figure 4 shows an example of a SIMULA program except that we use syntax in the style of C++, Java and C#. This example contains the following elements:

```
class Main:
{
    class Person:
        { name: text; age: integer; };
    class Set(size: integer):
        { rep: array(size);
          proc insert(e: object): { do ... };
          virtual proc display(): { do ... };
        };
    proc main():
        { Person Joe = new Person();
          Set S
          do S = new Set(99);
          S.insert(Joe);
          S.display()
        };
}
```

Figure 4 SIMULA-like program

- The class `Main` with local (nested) classes `Person` and `Set` and a procedure `main`.
- The class `Person` with instance variables `name` and `age`.
- The class `Set` with a parameter `size`, an instance variable (array) `rep` for representing the set, a non-virtual procedure (method) `insert`, and a virtual procedure `display`.
- The procedure `main` with reference variables `Joe` and `S`.

The example has the following characteristics:

- Person, Set and main are nested within class Main.
- Class instances (objects) are created by `new Set()`.
- Procedure instances are created by `S.insert(Joe)` and `S.display()`.

Figure 5 shows a snapshot of the execution of the program in Figure 4 at the point where `S.insert(Joe)` is executed at the end of `main`.

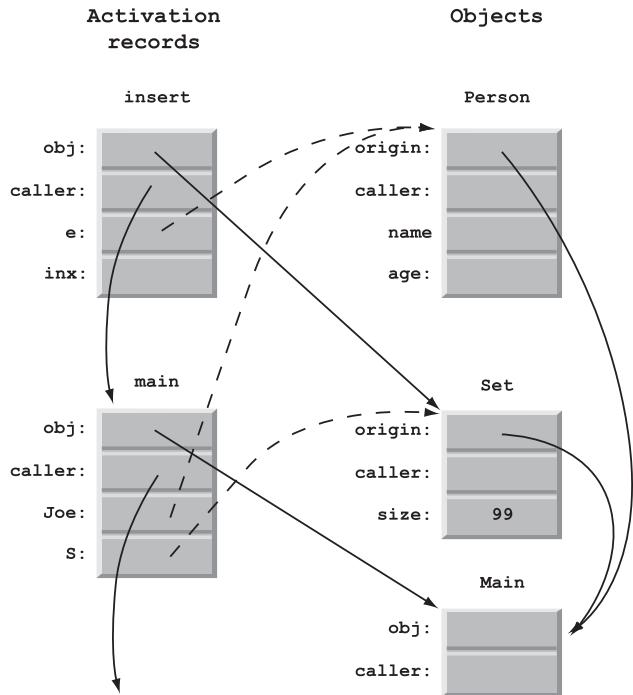


Figure 5 Objects and activation records

- The box named `main` is the activation record for `main`. It has a `caller` reference, which in this case is null since `main` is the first activation of this program. There is also an object reference (`obj`) to the enclosing `Main` object. In addition it has two data items `Joe` and `S` referring to instances of class `Set` and class `Person`
- The boxes named `Set` and `Person` are `Set` and `Person` objects respectively. Since the example is SIMULA-like, both objects have an `origin` for representing block structure and a `caller` representing the coroutine structure. For both objects `origin` refer to the enclosing `Main` object. The `caller` is null since `Set` and `Person` have no statement part.
- The box named `insert` is the activation record for the call of `S.insert(Joe)`. `Caller` refers to `main`. It has an object reference (`obj`) to the `Set` object on which the method is activated. In addition it has two instance variables `e` and `inx`. The variable `e` refers to the same object as `Joe`.

- The box named `Main` represents the `Main` object enclosing the `Set` and `Person` objects and the `main` activation record.

From the above presentation it should be clear that there is a strong structural similarity between an object and an activation record. The similarity is stronger for SIMULA than for languages like C++, Java and C#, since SIMULA has block structure and objects are coroutines. Technically one may think of a SIMULA class as a procedure where it is possible to obtain a reference to the activation record – the activation record is then an instance of the class.

5.1.4 The pattern

From the discussion of the similarities between class and procedure it follows that the following elements are candidates for a unified pattern:

- The name of the pattern
- The input parameters
- A possible superpattern
- Local data items
- Local procedures (methods) – virtual as well as non-virtual
- Local classes – possible nested classes
- A statement part – in the following called a do-part

One difference between a class and procedure is that a procedure may return a value, which is not the case for a class. To justify the unification we then had a minor challenge in defining the meaning of a return value for a pattern used as a class. We decided that we did not need an input parameter part for patterns. The rationale for this decision and the handling of return values are discussed in Section 5.8.1.

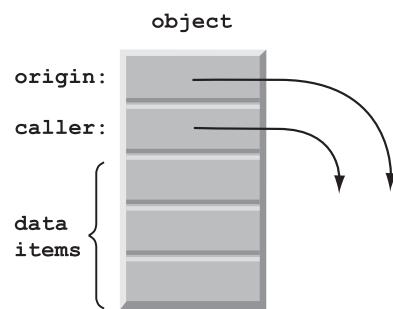


Figure 6 Object layout

In conclusion, we decided that a BETA object should have the layout shown in Figure 6. `Origin` represents the static link for nested procedures and objects and the object reference for method activations. Note that since a method is actually nested inside a class, there is no difference between the `origin` of a method activation and its object reference (`obj` in the above example). For patterns that are

not nested within other patterns, origin may be eliminated. Caller represents the dynamic link for activation records and coroutine objects. For patterns without a do-part, caller may be eliminated.

Figure 7 shows how the example from Figure 4 may be expressed in BETA. All of the classes and procedures have been expressed as patterns.

```
Main:
  (# Person: (# ... #);
  Set:
    (# insert:
      (# e: ^object
       enter e[] do ... #);
      display:< (# ... #);
      ...
    #);
  main:
    (# Joe: ^Person;
     S: ^Set
     do &Person[] -> Joe[];
     &Set[] -> S[];
     Joe[] -> S.insert;
     S.display
    #);
  #)
```

Figure 7 Pattern example

Basically it is a simple syntactic transformation:

- The keywords `class` and `proc` are removed.
- The brackets `{` and `}` are replaced by `(#` and `#)`.
- The `insert` input parameter part (`e: Object`) is replaced by a declaration of an object reference variable (`e: ^object`) and a specification that `e` is the input parameter (`enter e[]`).
- The symbol `^` in a declaration of a variable specifies that the variable is a (dynamic) reference – we show below that variables can also be defined as static.
- The symbol '`<`' specifies that `display` is a virtual pattern.
- The keyword '`do`' separates the declaration part and the do-part.
- The symbol `&` corresponds to `new` – i.e. `&Set` generates an instance of pattern `Set`.
- The symbol `[]` in an application of a name, as in `Joe[]`, signals that the value of `Joe[]` is a *reference* to the object `Joe`. This is in contrast to `Joe`, which has the *object* `Joe` as its value.
- Assignment has the form `exp -> v`, where the value of `exp` is assigned to `v`.
- The parentheses `()` are removed from procedure declarations and activations.

- The arrow `->` is also used for procedure arguments, which are treated as assignments. This is the case with `Joe[] -> S.insert`, where `Joe[]` is assigned to the input parameter `e[]`.

- Instances of a pattern may be created in two ways:

- The constructs `&Set[]`. The value of `&Set[]` is a reference to the new `Set` instance and corresponds to the new operator in most object-oriented languages.
- `S.display`. Here a new `display` instance is generated and executed (by executing its do-part). This corresponds to a procedure instance as shown in Figure 4.

The general form for a pattern is

```
P: superP           // super pattern
  (# A1; A2;...;An // attribute-part
   enter (V1, V2,...,Vs) // enter-part
   do I1; I2;...;Im // statements
   exit (R1, R2,...,Rt) // exit-part
  #)
```

- The super-part describes a possible super pattern.
- The attribute-part describes declaration of attributes including variables and local patterns
- The enter-part describes an optional list of input parameters.
- The do-part describes an optional list of executable statements.
- The exit-part describes an optional list of output values.

As is readily seen from this general form for a pattern, a pattern may define a simple record type (defining only attributes), it may define a class with methods (in which case the local patterns are methods), or it may define procedures/functions, in which case the enter/exit lists work as input/output parameters.

```
ia: S.insert;
ia = new S.insert();
ia.e = Joe;
ia.execute();
```

Figure 8 Decomposition of `S.insert(Joe)`

As mentioned above, a procedure call may be described as a generation of an activation record, transfer of arguments, and execution of the code of the procedure. In Figure 8, such a decomposition of the call `S.insert(Joe)` from Figure 4 is shown:

- A variable `ia` of type `S.insert` is declared.
- An instance of `S.insert()` is assigned to `ia`.
- The argument `e` is assigned the value of `Joe`.
- The statement part of `ia` is executed.

In BETA it is possible to write this code directly. This also implies that the activation record (object) ia may be executed several times by reapplying ia.execute(). Such procedure objects were referred to as static procedure instances and considered similar to FORTRAN subroutines.

Although in a pure version of BETA one could imagine that all procedure calls would be written as in Figure 8, this would obviously be too clumsy. From the beginning an abbreviation corresponding to a normal syntax for procedure call was introduced.

5.1.5 Subpatterns

With respect to subpatterns a number of issues were discussed over the years. As with a SIMULA subclass, a subpattern inherits all attributes of its superpattern. We did discuss the possibility of cancellation of attributes as in Eiffel, but found this to be incompatible with a modeling approach where a subpattern should represent a specialization of its superpattern. We also had to consider how to combine the enter-, do- and exit-parts of a pattern and its superpattern. For the enter- and exit-parts we decided on simple concatenation as in SIMULA, although alternatives were discussed, such as allowing an inner statement (cf. Section 5.6) inside an enter/exit part to specify where to put the enter/exit part of a given subpattern. The combination of do-parts is discussed in Section 5.6. The following is an example of a subpattern:

```
Student: Person (# ... #)
```

The pattern `Student` is defined as a subpattern of `Person`. The usual rules regarding name-based subtype substitutability applies for variables in BETA. As in most class-based languages, an instance of `Student` has all the properties defined in pattern `Person`.

In SIMULA a subclass can only be defined at the same block level as that in which its superclass is defined. The following example where `TT` is not defined at the same block level as its superclass `T` is therefore illegal:

```
class A: {                                // block-level 0
    class T: { ... }                      // block-level 1
    class X: { ... }                      // block-level 1
        class TT: T { ... } // block-level 2
    }
}
```

BETA does not have this restriction. The restriction in SIMULA was because of implementation problems. We return to this question in Section 6.4.

Multiple inheritance has been an issue since the days of SIMULA – we return to this issue in Section 5.5.1 and 5.8.12.

5.1.6 Modeling rationale

The rationale for one pattern as described in Section 3.1.2 and above is mainly technical. For a modeling language it

is essential to be able to represent concepts and phenomena of the application domain. Abstraction mechanisms like class, procedure and type may represent specialized concepts from the application domain. It seemed natural to be able to represent a concept in general. The idea of having one pattern mechanism generalizing all other abstraction mechanisms was then considered well motivated from this point of view also.

Since the primary purpose of patterns was to represent concepts, it has always been obvious that a subpattern should represent a specialized concept and thereby be a specialization of the superpattern. This implies that all properties of the superpattern are inherited by the subpattern. The ideal would be to ensure that a subpattern is always a behavioral specialization of the superpattern, but for good reasons it is not possible to ensure this by programming language mechanisms alone. The language rules were, however, designed to support behavioral specialization as much as possible.

The notions of type and class are closely associated. Programming language people with focus on the technical aspects of a language often use the term ‘type’, and the purpose of types is to improve the readability of a program, to make (static) type checking possible and to be able to generate efficient code. A type may, however, also represent a concept, and for BETA this was considered the main purpose of a type. Many researchers (like Pierre America [8] and William Cook [27]) think that classes and types should be distinguished – classes should only be used to construct objects and types should be used to define the interfaces of objects. We have always found that it was an unnecessary complication to distinguish between class and type.

There was also the issue of name or structural equivalence of types/classes. From a modeling point of view it is rarely questioned that name equivalence is the right choice. People in favor of structural equivalence seem to have a type-checking background. The names and types of attributes of different classes may coincidentally be the same, but the intention of the two classes might be quite different. Boris Magnusson [121] has given as example class `Cowboy` and class `Figure` that both may have a `draw` method. The meaning of `draw` for `Cowboy` is quite different from the meaning of `draw` for `Figure`. The name equivalence view is also consistent with the general view of a concept being defined by its name, intension (attributes) and extension (its instances).

Another issue that is constantly being discussed is whether or not a language should be statically or dynamically typed. From a modeling point of view there was never any doubt that BETA should be statically typed since the type (class) annotation of variables is an essential part of the description

of a model. BETA is, however, not completely statically typed – cf. Section 5.4.4 on co- and contravariance.

5.2 Singular objects

BETA supports singular objects directly and thereby avoids superfluous classes. The following declaration is an example of a singular object:

```
myKitchen: @ Room(# ... #)
```

The name of the object is myKitchen, and it has Room as a superpattern. The symbol @ specifies that an object is declared.⁸ The object is singular since the object-descriptor Room(# ... #) is given directly instead of a pattern like Kitchen.

Technically it is convenient to be able to describe an object without having to first declare a class and then instantiate an object – it is simply more compact.

With respect to modeling the rationale was as follows:

- When describing (modeling) real-life systems there are many examples of one-of-a-kind phenomena. The description of an apartment may contain various kinds of rooms, and since there are many instances of rooms it is quite natural to represent rooms as patterns and subpatterns (or classes and subclasses). An apartment usually also has a kitchen, and since most apartments have only one kitchen, the kitchen may be most naturally described as a singular object. It should be mentioned that any description (program) is made from a given perspective for a given purpose. In a description of one apartment it may be natural to describe the kitchen as a singular object, but in a more general description that involves apartments that may have more than one kitchen it may be more natural to include a kitchen pattern (or class).
- Development of system descriptions and programs is often evolutionary in the sense that the description evolves along with our understanding of the problem domain. During development it may be convenient to describe phenomena as singular objects; later in the process when more understanding is obtained the description is often refactored into patterns and objects. Technically it is easy to change the description of a singular object to a pattern and an instance – in the same way as a description of a singular phenomenon is easily generalized to be a description of a concept. For an elaboration of this see Chapter 18 in the BETA book [119].

Exploratory programming emphasizes the view of using objects in the exploratory phase, and it is the whole basis for prototype-based object-oriented programming as e.g. in

⁸ This is in contrast to ^, which specifies that a reference to an object is declared.

Self. However, as discussed by Madsen [113], prototype-based languages lack the possibility of restructuring objects into classes and objects when more knowledge of the domain has been obtained.

5.3 Block structure

Algol allowed nesting of blocks and procedures. SIMULA in addition allowed general nesting of classes, although there were some restrictions on the use of nested classes. For BETA it was quite natural that patterns and singular objects could be arbitrarily nested. Nesting of patterns comes almost by itself when there is no distinction between class and procedure. A pattern corresponding to a class with methods is defined as a class pattern containing procedure patterns, and the procedure patterns are nested inside the class pattern. The pattern Set in Figure 7 is an example – the patterns display and insert are nested inside the pattern Set. It is thus quite natural that patterns may be nested to an arbitrary level – just as procedures may be nested in Algol, Pascal and SIMULA, and classes may be nested in SIMULA. With nesting of patterns, nesting of singular objects comes naturally. A singular object may contain inner patterns, just as a pattern may contain inner singular objects.

A major distinction between Smalltalk and the SIMULA/BETA style of object-oriented programming is the lack of block-structure in Smalltalk. Since the mid-eighties, we have been trying to convince the object-oriented community of the advantages of block-structure, but with little success. In 1986 a paper with a number of examples of using block structure was presented at the Hawthorne Workshop (see Section 3.3) and also submitted to the first OOPSLA conference, but not accepted. It was later included in the book published as the result of the Hawthorne Workshop [111]. C++ (and later C#) does allow textual nesting of classes, but only to limit the scope of a given class. In C++ and C# a nested class cannot refer to variables and methods in the enclosing object. Block structure was added to Java in one of the first revisions, but there are a number of restrictions on the use of nested classes, which means that some of the generality is lost. As an example, it is possible to have classes nested within methods (local nested classes), but instances of these classes may not access nonfinal variables local to the method.

In Algol and SIMULA, the rationale for block structure was purely technical in the sense that it was very convenient to be able to nest procedures, classes and blocks. Block structure could be used to restrict the scope and lifetime of a given data item. For some time it was not at all obvious that block structure could be justified from a modeling point of view.

The first step towards a modeling justification for block structure was taken by Liskov and Zilles [109]. Here a

problem with defining classes representing grammars was presented. One of the elements of a grammar is its symbols. It is straightforward to define a class `Grammar` and a class `Symbol`. The problem pointed out by Liskov and Zilles was that the definition of class `Symbol` in their example was dependent on a given `Grammar`, i.e. symbols from an Algol grammar had different properties from symbols from a COBOL grammar. With a flat class structure it was complicated to define a class `Symbol` that was dependent on the actual grammar. With block structure it was straightforward to nest the `Symbol` class within the `Grammar` class.

Another example that helped clarify the modeling properties of block structure was the so-called *prototype abstraction relation problem* as formulated by Brian Smith [146]. Consider a model of a flight reservation system:

- Each entry in a flight schedule like SK471 describes a given flight by SAS from Copenhagen to Chicago leaving every day at 9:40 am with a scheduled flight time of 8 hours.
- A flight entry like SK471 might naturally be an instance of a class `FlightEntry`.
- Corresponding to a given flight entry there will be a number of actual flights taking place between Copenhagen and Chicago. One example is the flight on December 12, 2005 with an actual departure time of 9:45 and an actual flight time of 8 hours and 15 minutes. These actual flights might be modeled as instances of a class `SK471`.
- The dilemma is then that `SK471` may be represented as an instance of class `FlightEntry` or as a class `SK471`.
- With nested classes it is straightforward to define a class `FlightEntry` with an inner class `ActualFlight`. `SK471` may then be represented as an instance of `FlightEntry`. The `SK471` object will then contain a class `ActualFlight` that represents actual instances of flight `SK471`.

The grammar example and the prototype abstraction relation problem are discussed in Madsen's paper on block structure [111] and in the BETA book [119].

Eventually block structure ended up being conceived as a means for describing concepts and objects that depend on and are restricted to the lifetime of an enclosing object. In the BETA book [119], the term *localization* is used for this conceptual means. The modeling view is in fact consistent with the more technical view of block structure as a construct for restricting the lifetime of a given data item.

5.4 Virtual patterns

One of the implications of having just one abstraction mechanism was that we would need a virtual pattern mechanism in order to support virtual procedures. Since a

pattern may be used as e.g. a class, we needed to assure that it was meaningful to use a virtual pattern as a class. Algol, SIMULA and other languages had support for higher-order procedures and functions and proposals for higher-order types and classes had started to appear. Quite early in the BETA project it was noticed that there was a similarity between a procedure as a parameter and a virtual procedure. It was thus obvious to consider a unification of the two concepts. In the following we discuss virtual patterns used as virtual procedures and as virtual classes. Then we discuss parameterized classes and higher-order procedures and functions.

5.4.1 Virtual procedures

Virtual patterns may be used as virtual procedures, as illustrated by the pattern display in Figure 7. The main difference from virtual procedures in SIMULA and other languages is that in BETA a virtual procedure is not redefined in a subclass, but extended. The reason for this was a consequence of generalizing virtual procedure to cover virtual class, as described in the next section. Consider the example:

```
Person:
  (# name: @text;
   display:<
     (# do name[]->out.puttext; inner #)
   #)
Employee: Person
  (# salary: @integer;
   display:<(# do salary->out.putInt #
   #)
```

The `display` procedure in `Employee` is combined using `inner` with the one in `Person` yielding the following pattern

```
display:
  (# do name[] -> out.puttext;
   salary -> out.putInt
  #)
```

For further details, see the BETA book [119]; we return to this discussion in the sections on virtual class and specialization of actions.

Wegner and Zdonik [160] characterized the different notions of class/subclass relationships as name-, signature-, or behavior-compatible. SIMULA has signature equivalence since the signature of the method in the super- and subclass must be the same. This is not the case for Smalltalk since there are no types associated with the declaration of arguments. I.e. a method being redefined must have the same name and number of arguments as the one from the superclass, but the types may vary. For BETA it was obvious that at least the SIMULA rule should apply. The modeling emphasis of BETA implied that from a semantic point of view a subclass should be a specialization of the superclass – i.e. behaviorally compatible. This means that code executed by a redefined method should not break invariants established by the method in the superclass.

Behavioral equivalence cannot be guaranteed without a formal proof and therefore cannot be expressed as a language mechanism. For BETA we used the term structural compatibility as a stronger form than signature compatibility. In BETA it is not possible to eliminate code from the superclass. It is slightly closer to behavioral equivalence since it is guaranteed that a given sequence of code is always executed – but of course the effect of this can be undone in the subclass.

5.4.2 Virtual classes

As mentioned above, it was necessary to consider the implications of using a virtual pattern as a class. At a first glance, it was not clear that this would work, as illustrated by the following example:

```
Set:
  (# ElmType:< (# key: @integer #);
  newElement:
    (# S: ^ElmType;
    do &ElmType [] -> S[];
    newKey -> S.key;
    #)
  #);

PersonSet:
  Set(# ElmType::< (# name: @Text #))

PS: @PersonSet;
```

The pattern `Set` has a virtual pattern attribute `ElmType`, which is analogous to the virtual pattern attribute `display` of `Person` above. In `Person`, the pattern `display` is used as a procedure, whereas `ElmType` in `Set` is used as a class. In `newElement`, an instance of `ElmType` is created using `&ElmType[]`. This instance is assigned to the reference `S` and the attribute `key` of `S` is assigned to in `newKey -> S.key`.

In SIMULA a virtual procedure may be redefined in a subclass. If redefinition is also the semantics for a pattern used as a class then the `ElmType` in instances of `PersonSet` will be `ElmType` as defined in `PersonSet`. This implies that an execution of `&ElmType[]` in `PS.newElement` will create an instance of `ElmType` defined as `(# name: @Text #)`, and with no `key` attribute. A subsequent execution of `newKey -> S.key` will then break the type checking. This was considered incompatible with the type rules of SIMULA where at compile time it is possible to check that a remote access like `newKey -> S.key` is always safe.

We quickly realized that if `PersonSet.ElmType` was a subclass of `Set.ElmType`, then the type checking would not break, i.e. the declaration of `PersonSet` should be interpreted as:

```
PersonSet: Set
  (# ElmType::<
    Set.ElmType(# name: @Text #
  #))
```

That is, `ElmType` in `Set` is implicitly the superpattern of `ElmType` in `PersonSet`. We introduced the term *further binding* for this to distinguish the extension of a virtual from the traditional redefinition semantics of a virtual.

With redefinition of virtual patterns being replaced by the notion of extension, it was necessary to consider the implications of this for virtual patterns used as procedures. It did not take long to decide that extension was also useful for virtual patterns used as procedures. A very common style in object-oriented programming is that most methods being redefined start by executing the corresponding method in the superclass using `super`. With the extension semantics, one is always guaranteed that this is done. Furthermore, as discussed below in the section on specialization of actions, it is possible to execute code before and after code in the subclass.

As mentioned in Section 5.4.1, we assumed that signature compatibility from SIMULA should be carried over to BETA. The extension semantics includes this and in addition gives the stronger form of structural compatibility. Again, from a modeling point of view it was pretty obvious (to us) that this was the right choice.

The disadvantage of extension is less flexibility. With redefinition you may completely redefine the behavior of a class. One of the main differences between the U.S. school and Scandinavian school of object-orientation was that the U.S. school considered inheritance as a mechanism for incremental modification or reuse (sometimes called code sharing) [160]. It was considered important to construct a new class (a subclass) by inheriting as much as possible from one or more superclasses and just redefine properties that differ from those of the superclasses. Belonging to the Scandinavian school, we found it more important to support behavioral compatibility between subclasses than pure reuse.

The only situation we were not satisfied with was the case where a virtual procedure was defined as a default behavior and then later replaced by another procedure. This was quite common in SIMULA. We did consider introducing default bindings of virtuals, but if a default binding was specified, then it should not be possible to use information about the default binding. That is, if a virtual `v` is declared as `v:< A` and `AA` (a subpattern of `A`) is specified as the default binding, then `v` is only known to be an `A`. New attributes declared in `AA` cannot be accessed in instances of `v`. We did never implement this, but this form of default bindings was later included in SDL 92 (see Section 7.3).

5.4.3 Parameterized classes

It was a goal that virtual patterns should subsume higher-order parameter mechanisms like name and procedure parameters and traditional virtual procedures. In addition it was natural to consider using virtual patterns for defining

parameterized classes. The use of (locally defined) virtual patterns as described above was a step in the right direction: the pattern `PersonSet` may be used to represent sets of persons by their name, and other attributes like age and address may also be added to `ElmType`. We would, however, like to be able to insert objects of a pattern `Person` into a `PersonSet`. In order to do this we may define a parameterized class `Set` in the following way:

```
Set:
  (# ElmType:< (# #);
   insert:<
     (# x: ^object; e: ^ElmType
      enter x[]
      do &ElmType[] -> e[];
      inner;
      e[] -> add
    #)
  #)
```

The virtual pattern `ElmType` constitutes the type parameter of `Set`. The pattern `add` is assumed to store `e[]` in the representation of `Set`. A subclass of `Set` that may contain `Person` objects may be defined in the following way:

```
PersonSet: Set
  (# ElmType:::< (# P: ^Person #) #);
   insert:<(# do x[] -> e.P[] #)
  #)
```

The virtual pattern `ElmType` is extended to include a reference to a `Person`. The parameter `e[]` of `insert` is stored in `e.P[]`. This would work, but it is an indirect way to specify that `PersonSet` is a set of `Person` objects. Instead one would really like to write:

```
Set:
  (# ElmType:< object;
   insert:<
     (# x: ^ElmType
      enter x[]
      do (* add x[] to the rep. of Set *)
    #)
  #)

PersonSet: Set (# ElmType:::< Person #)
```

Here `ElmType` is declared as a virtual pattern of type `object`. In `PersonSet`, `ElmType` is extended to `Person`, and in this way, the declaration of `PersonSet` now clearly states that it is a set of `Person` objects. It turned out that it was quite straightforward to allow this form of semantics where a virtual in general can be qualified by and bound to a nonlocal pattern – just as a combination of local and nonlocal patterns would work. The general rule is that if a virtual pattern is declared as `T:< D` then `T` may be extended by `T:< D1` if `D1` is a subpattern of `D`. `T` may be further extended using `T:< D2` if `D2` is a subpattern of `D1`. The `PersonSet` above may be extended to a set holding `Students`, as in

```
StudentSet:
  PersonSet(# ElmType:::< Student #)
```

A *final binding* of the form `ElmType:: Student` may be used to specify that `ElmType` can no longer be extended.

Both forms (`v:< (# ... #)` and `v:< A`) of using virtual patterns have turned out to be useful in practice – examples may be found in the OOPSLA'89 paper [117], and the BETA book [119].

5.4.4 Co- and contravariance

For parameterized classes, static typing, subclass substitutability and co- and contravariance have been central issues. Most researchers seem to give static typing the highest priority, leading to – in our mind – limited and complicated proposals for supporting parameterized classes. In our 1990 OOPSLA paper [115] the handling of these issues in BETA was discussed. Subclass substitutability is of course a must, and covariance was considered more useful and natural than say contravariance. This implies that a limited form of run-time type checking is necessary when using parameterized classes – which in BETA are supported by patterns with virtual class patterns.

SIMULA, BETA, and other object-oriented languages do contain run-time type checking for so-called reverse assignment where a less qualified variable is assigned to a more qualified variable – like

```
aVehicle -> aBus
```

The run-time type checking necessary to handle covariance is similar to that needed for checking reverse assignment.

With emphasis on modeling it was quite obvious that covariance was preferred to contravariance, and it was needed for describing real-life systems. The supporters of contravariance seem mainly to be people with a static type-checking approach to programming.

It is often claimed in the literature (see e.g. [20]) that BETA is not type safe. This is because BETA requires some form of run-time type checking due to covariance. The compiler, however, gives a warning at all places where a run-time type check is inserted. It has often been discussed whether we should insist on an explicit cast in the program at all places where this run-time check is inserted. In SIMULA a reverse assignment may be written as

```
aBus :- aVehicle
```

In this case it is not clear from the program that an implicit cast is inserted by the compiler. SIMULA, however, also has explicit syntax for specifying that a cast is needed for a reverse assignment. It is possible to write

```
aBus:- aVehicle qua Bus
```

Here it is explicit that a cast is inserted. Introducing such an explicit syntax in BETA for reverse assignment and covariant parameters has often been discussed. As an afterthought, some of us would have preferred doing this from the beginning, since this would have ‘kept the static

typeziers away’⁹. However, whenever we suggested this to our users, they strongly objected to having to write an explicit cast. With respect to type safety it does not make any difference since a type error may still occur at run-time. We do, however, think that from a language design point of view it would be the right choice to insist on an explicit syntax, since it makes it clear that a run-time check is carried out.

With respect to static typing, it is pointed out in our OOPSLA’90 paper [115] that although the general use of virtual class patterns will involve run-time type checking, it is possible to avoid this by using final bindings and/or part objects (cf. Section 5.5). This has turned out to be very common in practice.

5.4.5 Higher-order procedures and functions

In many languages a procedure⁹ may be parameterized by procedures. A procedure specified as a parameter is called a *formal procedure*. The procedure passed as a parameter is called the *actual procedure*. It was an issue from the beginning of the project that formal procedures should be covered by the pattern concept – and it was quickly realized that this could be done by unifying the notions of virtual procedure and formal procedure.

Consider a procedure `fsum` parameterized by a formal procedure `f`, as in:

```
real proc fsum(real proc f){ ... }
```

An invocation of `fsum` may pass an actual procedure `sine` as in:

```
fsum(sine)
```

In BETA a formal procedure may be specified using a virtual procedure pattern as in:

```
fsum:(# f:< realFunction; ... #)
```

An invocation then corresponds to specifying a singular subpattern of `fsum` and a binding of `f` to the `sine` pattern:

```
fsum(# f:: sine #)
```

SIMULA inherited call-by-name parameters from Algol. Value parameters are well suited to pass values around – this is the case for simple values as well as references. Call-by-name-parameters, like formal procedures, involve execution of code. For a call-by-name parameter the actual parameter is evaluated every time the formal parameter is executed in the procedure body – this implies that the context of the procedure invocation must be passed (implicitly) as an argument. It was a goal to eliminate the need for call-by-name parameters, and the effect of call by name can in fact be obtained using virtual patterns.

⁹ In this section, *procedure* may be read as *procedure and/or function*.

5.4.6 Pattern variables

Virtual patterns partially support higher-order procedures in the sense that a virtual pattern may be considered a parameter of a given pattern. Originally BETA had no means for a pattern to return a pattern as a value. In general, virtual patterns do not make patterns first class values in the sense that they may be passed as arguments to procedures (through the enter part), returned as values (through the exit part) and be assigned to variables. For some years we thought that using virtual patterns as arguments fulfilled most needs to support higher-order procedures, although it was not as elegant as in functional languages.

Indirectly, the work of Ole Agesen, Svend Frølund and Michael H. Olsen on persistent objects for BETA [4, 5] made it evident that a more dynamic pattern concept was needed. When a persistent object is loaded, its class (pattern) may not be part of the program loading the object. There was thus a need to be able to load its associated pattern and assign it to some form of pattern variable. Agesen, Frølund, and Olsen suggested the notion of a *pattern variable*, which forms the basis for supporting patterns as first-class values.

Consider a pattern `Person` and subpatterns `Student` and `Employee`. A pattern variable `P` qualified by `Person` may be declared in the following way:

```
P: ## Person
```

`P` denotes a pattern that is either `Person` or some subpattern of `Person`. This is quite similar to a reference `R: ^Person` where `R` may refer to an instance of `Person` or subpattern of `Person`. The difference is that `R` denotes an object, whereas `P` denotes a pattern. `P` may be assigned a value in the following way:

```
Student## -> P##
```

`P` now denotes the pattern `Student` and an instantiation of `P` will generate an instance of `Student`. `P` may be assigned a new value as in:

```
Employee## -> P##
```

`P` now denotes the pattern `Employee` and an instantiation of `P` will result in an `Employee` object.

Pattern variables give full support to higher-order procedures in the sense that patterns may be passed as arguments to procedures, returned as values and assigned to variables.

5.5 Part objects

From the very start we distinguished between variables as references to autonomous objects separate from the referencing objects, and variables as part objects being constituents of a larger object. We had many examples where this distinction was obvious from a modeling point

of view: car objects with part objects body and wheels and references to a separate owner object, patient objects with organ part objects and a reference to a physician object, book objects with part objects (of type `Text`) representing the title and a reference to an author object, etc. In most of these examples there is always a question about perspective: for the owner, the car is not a car without four wheel part objects, while a mechanic has no problem with cars in which the wheels are separate (i.e. not part) objects.

We were not alone in thinking that from a modeling point of view it is obvious that (physical) objects consist of parts. At the ECOOP'87 conference Blake and Cook presented a paper on introducing part objects on top of Smalltalk [12]. In [163] Kasper Østerbye described the proper (and combined) use of ‘parts, wholes and subclasses’. The example we used in our paper on part objects [118] was inspired by an example from Booch [14]: the problem presented there was to represent (in Smalltalk) buildings (for the purpose of heating control) as objects consisting of objects representing the parts of the building. While the Booch method and notation had no problem in modeling this, it was not possible in Smalltalk, where only references were supported.

In our paper we used an apartment with kitchen, bath, etc. as example:

```
Apartment:
  (# theKitchen: @Kitchen;
   theBathroom: @Bathroom;
   theBedroom: @Bedroom;
   theFamilyRoom: @FamilyRoom;
   theOwner: ^Person;
   theAddress: @Address;
   ...
  #)
```

Note the difference between the rooms of the apartment modeled by part objects (using `@`) and the owner modeled by a *reference variable* (`theOwner`) to a separate object (using `^`).

Although BETA was designed from a modeling point of view, it was still a programming language, so we did not distinguish between parts objects modeling real parts (as the rooms above) and part objects implementing a property (`theAddress` property above) – in BETA terms they are all part objects.

Another problem with Smalltalk was that it allowed external access only to methods, while all instance variables were regarded as private. The example would in Smalltalk have to have access methods for all rooms, and in order to get to the properties of these rooms, one would have to do this via these access methods. In BETA we allowed access to variables (both part objects and references) directly, so with the example above it is possible to e.g. invoke the `paint` method in `theKitchen` as follows:

```
...; myApartment.theKitchen.paint; ...
```

Comparing BETA with Java, a reference to an object (like `theOwner` variable above) corresponds to a Java reference variable typed with `Person`, while a part object is a final reference variable.

A less important rationale for part objects was that part objects reflected the way ordinary variables of predefined value types like `Integer`, `Real`, `Boolean`, etc. were implemented, and we regarded e.g. `Integer`, `Real` and `Boolean` as (predefined) patterns.

5.5.1 Inheritance from part objects

In the part object paper we wrote:

‘In addition to the obvious purpose of modeling that wholes consist of parts, part objects may also be used to model that the containing object is characterized by various aspects,¹⁰ where these aspects are defined by other classes.’

This reflects discussions we had, but they never led to additional language concepts. It does, however, illustrate the power of combining part objects, block structure and virtual patterns.

Multiple inheritance by part objects. We explored the possibility of using part objects to represent various aspects of a concept. This was partially done in order to provide alternatives to multiple inheritance (see also Section 5.8.12). In the following we give an example of using part objects to represent aspects.

Persons and Companies are examples of objects that may be characterized by aspects such as being addressable and taxable. The aspect of being addressable may be represented by the pattern:

```
Addressable:
  (# street: @StreetName;
   ...
   printLabel:< (# ... #);
   sendMail:< (# ... #)
  #)
```

Similarly, a taxable aspect may be represented by:

```
Taxable:
  (# income: @integer;
   ...
   makeTaxReturn: < (# ... #);
   pay:< (# do ... #)
  #)
```

A pattern `Person` characterized by being addressable and taxable may then be described as follows:

¹⁰ Here *aspect* is used as a general term and does not refer to *aspect-oriented programming*.

```

Person:
  (# name: @PersonName;
   myAddr: @Addressable
     (# printLabel::<
      (# do ...;name.print;... #);
      sendMail::< (# ... #)
     #);
   myTaxable: Taxable
     (# makeTaxReturn::< (# ... #);
      pay::< (# ... #)
     #)
  #)

```

As the descriptor of the `myAddr` part object has `Addressable` as a superpattern, the `printLabel` and `sendMail` virtuals can be extended¹¹. Since these extensions are nested within pattern `Person`, an attribute like `Name` is visible. This implies that it is possible to extend `printLabel` and `sendMail` to be specific for `Person`.

A pattern `Company` may be defined in a similar way:

```

Company:
  (# name: @CompanyName;
   logo: @Picture;
   myAddr:@Addressable
     (# printLabel::<
      (# ...;
       name.print;
       logo.print; ...
      #);
     sendMail::< (# ... #)
    #);
   myTaxable: Taxable(# ... #)
  #)

```

Again, notice that a virtual binding like `printLabel` may refer to attributes of the enclosing `Company` object.

In languages with multiple inheritance, `Person` may be defined as inheriting from `Addressable` and `Taxable`. From a modeling point of view we found it doubtful to define say `Person` as a subclass of `Addressable` and `Taxable`. From a technical point of view the binding of virtuals of `Addressable` and `Taxable` in `Person` will all appear at the same level when using multiple inheritance. Using part objects these will be grouped logically. A disadvantage is that these virtuals have to be denoted via the part object, as in

```

aPerson.myAddr.printLabel
aCompany.myTaxable.pay

```

The advantage is that the possibility of name conflicts does not arise.

5.5.2 References to part objects

Subtype substitutability is a key property of object-oriented languages: if e.g. `Bus` is a subclass of `Vehicle` then a reference of type `Vehicle` may refer to instances of class

¹¹ It is not important that extension semantics be used – the same technique may be used with redefinition of virtuals.

`Bus`. For an aspect like `Addressable` there is not a class/subclass relationship with e.g. class `Person`. If multiple inheritance is used to make `Person` inherit from `Addressable` then a reference of type `Addressable` may refer to instances of class `Person`.

In BETA it is possible to obtain a reference to a part object. This means that a reference of type `Addressable` may refer to a part object of type `Addressable` embedded within a `Person` object. If `anAddr1` and `anAddr2` are of type `Addressable` then the statements below will imply that `anAddr1` and `anAddr2` will refer the `Addressable` part-object of `aPerson` and `aCompany` respectively:

```

aPerson.myAddr[] -> anAddr1[];
aCompany.myAddr[] -> anAddr2[];

```

The effect of this is that `anAddr1` and `anAddr2` refer indirectly to a `Person` and a `Company` object, respectively. This is analogous to a reference of type `Vehicle` may refer to an instance of class `Bus`. It is thus possible to have code that handles `Addressable` objects independently of whether the `Addressable` objects inherits from `Addressable` or have `Addressable` as a part object: Suppose that we have defined `Company` as a subpattern of `Addressable` and `Person` containing an `Addressable` part object as shown above. We may then assign to `anAddr1` and `anAddr2` as follows (assuming that `anAddr1` and `anAddr2` are of type `Addressable`):

```

aCompany[] -> anAddr1
aPerson.myAddr[] -> anAddr2[]

```

Figure 9a shows how `anAddr1` may refer to a `Company`-object as a subpattern of `Addressable`. Figure 9b shows how `anAddr2` may refer to an `Addressable` part object of a `Person` object.

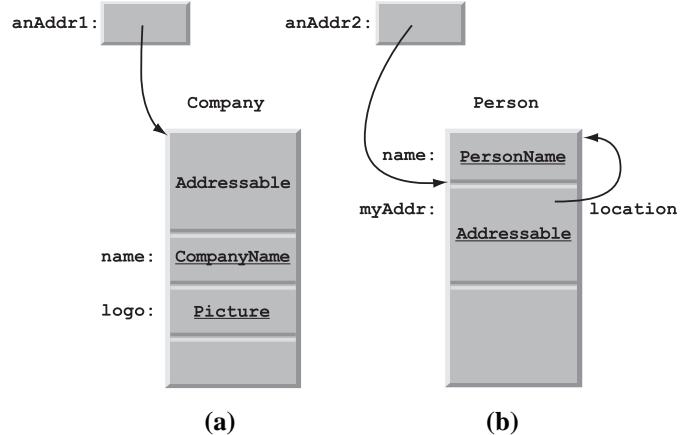


Figure 9 Inheritance from `Addressable` as super and as part object

A procedure handling `Addressable` objects – like calling `PrintLabel` – may then be called with `anAddr1` or `anAddr2` as its argument.

In many object-oriented languages it is also possible to make a reverse assignment (sometimes called casting) like

```
aVehicle[] -> aBus[]
```

Since it cannot be statically determined if `aVehicle` actually refers an instance of class `Bus`, a run-time type check is needed.

In order to be able to do a similar thing for part objects, we proposed in our part-object paper that a part object be given an extra *location* field containing a reference to the containing object. That is the `myAddr` part of a `Person` object is referencing the containing `Person` object. It is then possible to make a reverse assignment of the form

```
anAddr1.location[] -> aPerson[]
```

As for the normal case of reverse assignment, a run-time check must be inserted in order to check that `anAddr1` is actually a part of a `Person`. After publication of the part object paper, we realized that it would be possible to use the syntax

```
anAddr1[] -> aPerson[]
```

and extend the runtime check to check whether the object referred by `anAddr1` is a subclass of `Person` or a part object of `Person`.

In Figure 9b, the *location* field of the Addressable part object is shown. The concept of *location* was experimentally implemented, but did not become part of the released implementations.

5.6 Specialization of actions

From the very beginning we had the approach that specialization should apply to all aspects of a pattern, i.e. it should also be possible to specialize the behavior part of a pattern, not only types of attributes and local patterns. The inspiration was the inner mechanism of SIMULA. A class in SIMULA has an action part, and the inner mechanism allows the combination of the action parts of superclass and subclass. However, we had to generalize the SIMULA inner. In SIMULA, inner was simply a means for syntactically splitting the class body in two. The body of a subclass was defined to be a (textual) concatenation of the pre-inner body part of the superclass, the body part of the subclass, and the post-inner body part of the superclass. In BETA we rather defined inner as a special imperative that – when executed by the superpattern code – implied an execution of the subpattern do-part. This implied that an inner may appear at any place where a statement may appear, be executed several times (e.g. in a loop) and that an action part may contain more than one inner.

5.6.1 Inner also for method patterns

The fact that inner was defined for patterns in general and not only for classes as in SIMULA implied that it was useful also for patterns that defined methods. It was thereby

possible to define the general behavior of e.g. the method pattern `Open` of class `File`, with bookkeeping behavior before and after inner, use this general `Open` as a superpattern for `OpenRead` and `OpenWrite`, adding behavior needed for these, and finally have user-defined method patterns specializing `OpenRead` and `OpenWrite` for specific types of files.

Independently of this, Jean Vaucher had developed the same idea but applied to procedures in SIMULA [157].

5.6.2 Control structures and iterators

As mentioned in Section 5.1.1, it was a goal for BETA that it should be possible to define control structures by means of patterns. A simple example of the use of inner for defining control structures is the following pattern:

```
cycle: (# do inner; restart cycle #)
```

Given two file objects `F` and `G`, the following code simply copies `F` to `G`:

```
L: cycle(#  
      do (if F.eos then (* end-of-stream *)  
           leave L  
           if);  
           F.get -> G.put  
      #);
```

This is done by giving the copying code as the main do-part of an object being a specialization of `cycle`. The copying code will be executed for each execution of `inner` in the superpattern `cycle`.

The perhaps most striking example of the use of inner for defining control structure abstractions is the ability to define iterators on collection objects. If `mySet` is an instance of a pattern `Set` then the elements of the `mySet` may be iterated over by

```
mySet.scan(# do current.display #)
```

The variable `current` is defined in `scan` and refers to the current element of the set. The superpattern `mySet.scan` is an example of a remote pattern used as a superpattern. This is an example of another generalization of SIMULA.

Someone¹² has suggested that a data abstraction should define its associated control structures. By using patterns and inner it is possible in BETA to define control structures associated with any given data structure. Other languages had this kind of mechanism as built-in mechanisms for built-in data structures, while we could define this for any kind of user-defined data structure. At this time in the development there were languages with all kinds of fancy control structures (variations over while and for statements). We refrained from doing this, as it was

¹² We think this was suggested by Hoare, but have been unable to find a reference.

possible to define these by a combination of inner and virtual patterns.

The basic built-in control structures are `leave` and `restart`, which are restricted forms of the `goto` statement, and the conditional `if` statement. We were much influenced by the strong focus on structured programming in the seventies. Dijkstra published his influential paper on ‘`goto` considered harmful’ [36]. `Leave` and `restart` have the property that they can only jump to labels that are visible in the enclosing scope, i.e. continue execution either at the end of the current block or at the beginning at an enclosing “block”. In addition the corresponding control graphs have the property of being reducible. In another influential paper on guarded commands [37], Dijkstra suggested nondeterministic `if` and `while` statements. In addition, there was no `else` clause since Dijkstra argued that the programmer should explicitly list all possible cases. We found his argument quite convincing and as a consequence the BETA `if`-statement was originally nondeterministic and had no `else` clause. However, any reasonable implementation of an `if` statement would test the various cases in some fixed order and our experience is that the programmer quickly relies on this – this means that the program may be executed differently if a different compiler is used. It is of course important to distinguish the language definition from a concrete implementation, but in this case it just seems to add another source of errors. In addition it is quite inconvenient in practice not to have an `else` clause. We thus changed the `if` statement to be deterministic and added an `else` clause.

In principle we could have relied on `leave/restart` statements and `if` statements, but also a `for` statement was added. It is, however, quite simple to define a `for` statement as an abstraction using the existing basic control structures. However, the syntax for using control abstractions was not elegant – in fact Jørgen Lindskov Knudsen once said that it was clumsy. Today we may agree, and e.g. Smalltalk has a much more elegant syntax with respect to these matters. In the beginning of the BETA project we assumed that there would be (as an elegant and natural way to overcome these kinds of inconveniences) a distinction between basic BETA and standard BETA where the latter was an extension of basic BETA with special syntax for common abstractions. This distinction was inspired by SIMULA, which has special syntax for some abstractions defined in `class Simulation`.

Furthermore, we considered special syntax for `while` and `repeat` as in Pascal, but this was never included. The `for` statement may be seen as reminiscent of such special syntax.

5.6.3 Modeling

The phenomena of a given application domain include physical material (represented by objects), measurable

properties (represented by values of attributes) and transformations (represented by actions) of properties of the physical material – cf. Section 4.2. The traditional class/subclass mechanisms were useful for representing classification hierarchies on physical material. From a modeling point of view it was just as necessary to represent classification hierarchies of actions. This guided the design of using the inner mechanism to combine action parts and thereby be able to represent a classification hierarchy of methods and/or concurrent processes. The paper *Classification of Actions or Inheritance Also for Methods* [101] is an account of this.

5.7 Dynamic structure

From the beginning it was quite clear that BETA should be a concurrent object-oriented programming language. This was motivated from a technical as well as a modeling point of view. In the seventies there was lot of research activity in concurrent programming. Most of the literature on concurrency was quite technical and we spent a lot of time analyzing the different forms of concurrency in computer systems and languages. This led to the following classification of concurrency:

- **Hidden concurrency** is where concurrent execution of the code is an optimization made by a compiler – e.g. concurrent execution of independent parts of an expression.
- **Exploited concurrency** is where concurrency is used explicitly by a programmer to implement an efficient algorithm – concurrent sorting algorithms are examples of this.
- **Inherent concurrency** is where the program executes in an environment with concurrent nodes – typically a distributed system with several nodes.

We felt that it was necessary to clarify such conceptual issues in order to design programming language mechanisms. With the emphasis on modeling it was quite clear that *inherent concurrency* should be the primary target of concurrency in BETA.

The quasi-parallel system concept of SIMULA was the starting point for designing the dynamic structure of BETA systems. As mentioned in Section 3.1.3, quasi-parallel systems in SIMULA are based on coroutines, but the SIMULA coroutine mechanism did not support full concurrency and is furthermore quite complex. Conceptually, the SIMULA coroutine mechanism appears simple and elegant, but certain technical details are quite complicated. The coroutine system described by Dahl and Hoare in their famous book on structured programming [29] is a simplified version of the SIMULA coroutine system.

SIMULA did not have mechanisms for communication and synchronization, but several research results within

concurrent programming languages were published in the seventies. Concurrent Pascal [17] was a major milestone with regard to programming languages for concurrent programming. Concurrent Pascal was built upon the SIMULA class concept, but the class concept was specialized into three variants, class, process and monitor. This was sort of the opposite of the BETA goal of unification of concepts. In addition Concurrent Pascal did not have subclasses and virtual procedures. The monitor construct suggested by Brinch-Hansen and Hoare [57] has proved its usability in practice, and was an obvious candidate for inclusion in BETA. However, a number of problems with the monitor concept were recognized and several papers on alternative mechanisms were published by Brinch-Hansen and others [18].

Another important research milestone was CSP [58] where communication and synchronization was handled by input and output commands. Nondeterministic guarded commands were used for selecting input from other processes. We were very much influenced by CSP and later Ada [2] with respect to the design of communication and synchronization in BETA. In Ada communication and synchronization were based on the rendezvous mechanism, which is similar to input/output commands except that procedure calls are used instead of output commands.

5.7.1 The first version

From the beginning, a BETA system was considered a collection of coroutines possibly executing in parallel. Each coroutine is organized as a stack of objects corresponding to the stack of activation records. A coroutine is thus a simple form of thread. In the nonconcurrent situation, at most one coroutine is executing at a given point in time. Since activation records in BETA are subsumed by objects, the activation records may be instances of patterns or singular objects.

The SIMULA coroutine mechanism was quite well understood and the main work of designing coroutines for BETA was to simplify the SIMULA mechanism. SIMULA has symmetric as well as asymmetric coroutines [30]. In BETA there are only asymmetric coroutines – a symmetric coroutine system can be defined as an abstraction. In BETA it is furthermore possible to transfer parameters when a coroutine is called.

Conceptually it was pretty straightforward to imagine BETA coroutines executing concurrently. It was much harder to design mechanisms for communication and synchronization and this part went through several iterations.

The first published approach to communication and synchronization in BETA was based on the CSP/Ada rendezvous mechanism, mainly in the Ada style since procedure calls were used for communication. From a

modeling point of view this seemed a good choice since the rendezvous mechanism allowed direct communication between concurrent processes. With monitors all communication was indirect – of course, this may also be justified from a modeling point of view. However, since monitors could be simulated using processes and rendezvous we found that we had a solution that could support both forms of communication between processes.

In CSP and Ada input and output commands are not symmetric: input-commands (accept statements) may be used only in a guarded command. The possibility of allowing output commands as guards in CSP is mentioned by Hoare [58]. For BETA we considered it essential to allow a symmetric use of input and output commands in guarded commands. We also found guarded commands inexpedient for modeling a process engaged in (nondeterministic) communication with two or more processes. Below we give an example of this.

The following example is typical of the programming style used with guarded commands:

- Consider a process Q engaged in communication with two other processes P1 and P2.
- Q is engaged in the following sequential process with P1


```
Q1: cycle{P1.get(V1); S1; P1.put(e1); S2}
```

Q gets a value from P1, does some processing, sends a value to P1 and does some further processing. Note that rendezvous semantics is assumed for method invocations. This means that Q1 may have to wait at e.g. P1.get(V1) until P1 accepts the call.
- Q is also engaged in the following sequential process with P2:


```
Q2:cycle{ P2.put(e2); S3; P2.get(V2); S4 }
```
- Q1 and Q2 may access variables in Q. A solution where Q1 and Q2 are executed concurrently as in:


```
Q: { ... do (Q1 || Q2) }
```

where || means concurrency will therefore not work unless access to variables in Q is synchronized. And this is not what we want – in general we want to support cooperative scheduling at the language level.
- The two sequential processes have to be interleaved in some way to guarantee mutual access to variables in Q. It is not acceptable to wait for P1 if P2 is ready to communicate or vice versa. For instance, when waiting for P1.get one will have to place a guard that in addition accepts P2.put or P2.get. It is, however, difficult to retain the sequentiality between P1.get and P1.put and between P2.put and P2.get. Robin Milner

proposed the following solution using Boolean variables¹³:

```

Q:
{... do
  if
    B1 and P1.get(V1) then S1;B1:= false
    not B1 and P1.put(e1) then S2;B1:= true
    B2 and P2.put(e2) then S3;B2:= false
    not B2 and P2.get(V2) then S4;B2:= true
  fi
}

```

From a programming as well as a modeling point of view, we found this programming style problematic since the two sequential processes Q_1 and Q_2 are implicit. We did think that this was a step backward since Q_1 and Q_2 were much better implemented as coroutines. One might consider executing Q_1 and Q_2 in parallel but this would imply that Q_1 and Q_2 must synchronize access to shared data. This would add overhead and extra code to this example.

Eventually we arrived at the notion of *alternation*, which allows an active object to execute two or more routines while at most one at a time is actually executing. The above example would then look like

```

Q:
{...
 Q1: alternatingTask
  {cycle{ P1.get(V1);S1;P1.put(e1);S2}
  Q2: alternatingTask
  {cycle{ P2.put(e2);S3;P2.get(V2);S4}
 do (Q1 | Q2)
}

```

The statement $(Q_1 \mid Q_2)$ implies that Q_1 and Q_2 are executed in alternation. Execution of Q_1 and Q_2 may interleave at the communication points. At a given point in time Q_1 may be waiting at say $P1.put(e1)$ and Q_2 at $P2.get(v2)$. If $P1$ is ready to communicate, then Q_1 may be resumed. If on the other hand $P2$ is ready before $P1$ then Q_2 may be resumed.

The statement $(Q_1 \mid Q_2)$ is similar to $(Q_1 \parallel Q_2)$ – the former means alternation (interleaved execution at well defined points) and the latter means concurrent execution. Note, that the example is not expressed in BETA, whose syntax is slightly more complicated.

The version of BETA based on CSP/Ada-like rendezvous and with support for alternation is described in our paper entitled Multisequential Execution in the BETA Programming Language [97] (also published in [145]).

5.7.2 The final version

We were happy with the generalized rendezvous mechanism – it seemed simple and general, But when we

¹³ The syntax is CSP/Ada-like,

started using and implementing it, we discovered a number of problems:

- Although the rendezvous mechanism can be used to simulate monitors it turned out to be pretty awkward in practice. As mentioned above the monitor is one of the few concurrency abstractions that have proved to be useful in practice.
- It turned out to be inherently complicated to implement symmetric guarded commands – at least we were not able to come up with a satisfactory solution. In [70] an implementation was proposed, but it was quite complicated.

In addition we realized that the technique for defining a monitor abstraction as presented by Jean Vaucher [157] could also be used to define a rendezvous abstraction, alternation and several other types of concurrency abstractions including semi-coroutines in the style of SIMULA, and alternation. In late 1990 and early 1991, a major revision of the mechanisms for communication and synchronization was made. As of today, BETA has the following mechanisms:

- The basic primitive for synchronization in BETA is the *semaphore*.
- Higher-order concurrency abstractions such as monitor, and Ada-like rendezvous, and a number of other concurrency abstractions are defined by means of patterns in the Mjølner BETA libraries. The generality of the pattern concept, the inner mechanism and virtual patterns are essential for doing this. Wolfgang Kreutzer and Kasper Østerbye [80, 165] also defined their own concurrency abstractions.
- In BETA it is possible to define cooperative as well as preemptive (hierarchical) schedulers in the style of SIMULA. Although there were other languages that allowed implementation of schedulers, they were in our opinion pretty ad hoc and not as elegant and general as in SIMULA. At that time and even today, there does not seem to be just one way of scheduling processes.

For details about coroutines, concurrency, synchronization, and scheduling see the BETA book [119].

5.7.3 Modeling

From a modeling perspective there was obviously a need for full concurrency. The real world consists of active agents carrying out actions concurrently.

In DELTA it is possible to specify concurrent objects, but since DELTA is for system description and not programming, the DELTA concepts were not transferable to a programming language. To understand concurrency from a technical as well as a modeling point of view, we engaged in a number of studies of models for concurrency especially based on Petri nets. One result of this was the

language Epsilon [65], which was a subset of DELTA formalized by a Petri net model.

For coroutines it was not obvious that they could be justified from a modeling perspective. The notion of *alternation* was derived in order to have a conceptual understanding of coroutines from a modeling point of view. An agent in a travel agency may be engaged in several (alternating) activities like ‘tour planning’, ‘customer service’ and ‘invoicing’. At a given point in time the agent will be carrying out at most one of these activities.

As for coroutines, the notion of scheduling was not immediately obvious from a modeling point of view. This, however, led to the notion of an ensemble as described in Section 5.8.7 below.

5.8 Other issues

Here we discuss some of the other language elements that were considered for BETA. This includes language constructs that were discussed but not included in BETA.

5.8.1 Parameters and return values

In block-structured languages like Algol, the parameters of a procedure define an implicit block level:

```
foo(a,b,c: integer) { x,y,z: real do ... }
```

Here the parameters *a,b,c* corresponds to a block level and the local variables *x,y,z* are at an inner block level. For BETA the goal was that the implicit block level defined by the parameters should be explicit. A procedure pattern like *foo* should then be defined as follows:

```
foo:
  (# a,b,c: integer
  do (# x,y,z: integer do ... #)
  #)
```

The parameters are defined as data items at the outermost level, and the local variables are defined in a singular object in the *do* part.

With respect to return values, the initial design was to follow the Algol style and define a return value for a procedure – which in fact is still the style used in most mainstream languages. In most languages a procedure may also return values using call-by-reference and/or call-by-name parameters. However, many researchers considered it bad style to write a procedure that returns values through both its parameters and its return value. This style was (and still is), however, often used if a procedure needs to return more than one value. For BETA (as mentioned elsewhere), call-by-name was not an issue since it was subsumed by virtual patterns. As mentioned below, we did find that call-by-reference parameters would blur the distinction between values and objects. There were language proposals suggesting call-by-return as an alternative to call-by-reference. The advantage of call-by-return was that the actual parameter did not change during the execution of the

procedure, but was first changed when the procedure terminated. We did find a need to be able to return more than one value from a procedure and in some languages (like Ada) a variable could be marked as *in*, *out* or *inout* corresponding to call-by-value, -return or both. Finally, there was also a discussion on whether or not arguments should be passed by position or by the name of the parameter. In the first version of BETA all data items at the outermost level could be used as arguments and/or return values, and the name of a data item was used to pass arguments and return values. The pattern *foo* above might then be invoked as follows:

```
foo(put a:=e1, b:=e2) (get v:=b, w:=c)
```

We later found this too verbose, and position-based parameters were introduced in the form of enter/exit lists. The pattern *foo* would then be declared as follows:

```
foo: (# a,b,c: integer
      enter (a,b) do (# ... #)
      exit (b,c)
      #)
```

and invoked as follows:

```
(e1,e2) -> foo -> (v,w)
```

In this example, *enter* corresponds to defining *a,b* as *in* parameters and *exit* corresponds to defining *b,c* as *out* parameters, i.e. *b* was in fact an *inout* parameter.

There were a number of intermediate steps before the enter/exit parts were introduced in their present form. One step was replacing the traditional syntax for calling a procedure with the above (and current) postfix notation. In a traditional syntax the above call would look like:

```
(v,w) := foo(e1,e2)
```

If *e1* and *e2* also were calls to functions, a traditional call might look like:

```
(v,w) := foo(bar(f1,f2),fisk(g1,g2))
```

We did not find this to be the best syntax with respect to readability – in addition, we would like to write code as close as possible to the order of execution. This then led to the postfix notation where the above call will be written as

```
(f1,f2)->bar,(g1,g2)->fisk)->foo->(v,w)
```

We found this more readable, but others may of course disagree.

The enter/exit part may be used to define value types. In this case, the *exit* part defines the value of the object and the *enter* part defines assignment (or enforcement) of a new value on the object. The following example shows the definition of a complex number:

```
complex:
  (# x,y: @ real enter(x,y) exit(x,y) #)
```

Complex variables may be defined as follows:

```
C1,C2: @complex
```

They may be assigned and compared: In $c1 \rightarrow c2$, the exit part of $c1$ is assigned to the enter part of $c2$. In $c1 = c2$ the exit part of $c1$ is compared to the exit part of $c2$.

As part of defining a value type we would also like code to be associated with the value and assignment. For this reason, the enter/exit-part is actually a list of evaluations that may contain code to be executed. For purely illustrative purposes the following definition of complex keeps track of the number of times the value is read or assigned:

```
complex:  
  (# x,y: @real; n,m: @integer  
   enter (# enter(x,y) do n+1 -> n #)  
   exit (# do m+1 -> m exit (m,y) #)  
  #)
```

Complex may also have a do-part, which is executed whenever enter or exit is executed. If $c1$ is a complex object with a do-part then

- In $c1 \rightarrow E$, the do, and exit part of $c1$ is executed
- In $E \rightarrow c1$, the enter- and do part of $c1$ is executed
- In $E \rightarrow c1 \rightarrow F$, the enter, do and exit parts of $c1$ are executed.

The do part is thus executed whenever an object is accessed, the enter part when it is assigned and the exit part when the value is fetched.

One problem with the above definitions of complex is that the representation of the value is exposed. It is possible to assign simple values and decompose the exit part, as in

```
(3.14,1.11) -> C1 -> (q,w)
```

To prevent this, it was once part of the language that one could restrict the type of values that could be assigned/read:

```
complex:  
  (# x,y: @real  
   from complex enter(x,y)  
   to complex exit(x,y)  
  #)
```

In general any pattern could be written after from/to, but there was never any use of this generality and since we never became really happy with using enter/exit to define value types, the from/to-parts were abandoned.

The SIMULA assignment operators $:=$ and $:=$ were taken over for BETA. In the beginning $=>$ was used for assignment of values and $@>$ for assignment of references. However, since enter/exit-lists and lists in general may contain a mixture of values and references, we either had to introduce a third assignment operator to be used for such a mixture, or use one operator. Eventually \rightarrow was selected. The distinction between value and object is thus no longer explicit in the assignment operator. Instead, this is expressed by means of $[]$. An expression $x[]$ denotes the

reference to the object referred by x . An expression x denotes the value of the object.

5.8.2 Value concept

The distinction between object and value has been important for the design of BETA. This is yet another example of the influence of SIMULA as exemplified through the operators $:=$ and $:=$. In the previous section, we have described how enter/exit may be used to define value types. In this section we discuss some of the design considerations regarding the value concept.

As mentioned, the SIMULA class construct was a major inspiration for the notion of abstract data types developed in the seventies. For Nygaard a data type was an abstraction for defining values, and he found that the use of the class concept for this purpose might create conceptual confusion. In SIMULA, Dahl and Nygaard tried to introduce a concept of value types at a very late stage, but some of the main partners developing the SIMULA compilers refused to accept a major change at that late point of the project. The notion of value type was further discussed in the DELTA project and, as mentioned in Section 2.4, was one of the subprojects defined in JLP. Naturally the concept of value types was carried over to the BETA project.

One may ask why it should be necessary to distinguish value types from classes – why are values not just instances of classes? The distinction between object and value is not explicit in mainstream object-oriented languages. In Smalltalk values are immutable objects. In C++, Java and C# values are not objects, but there does not seem to be a conceptual distinction between object and value – the distinction seems mainly to be motivated by efficiency considerations.

From a modeling point of view, it is quite important to be able to distinguish between values and objects. As mentioned in Section 4, values represent measurable properties of objects. In 1982 MacLennan [110] formulated the distinction in the following way:

... values are abstractions, and hence atemporal, unchangeable, and non-instantiated. We have shown that objects correspond to real world entities, and hence exist in time, are changeable, have state, and are instantiated, and can be created, destroyed and shared. These concepts are implicit in most programming languages, but are not well delimited.

One implication of the distinction between value and object was that support for references to values as known from Algol 68 and C was ruled out from the beginning. A variable in BETA either holds a value or a reference to an object.

Another implication was that a value conceptually cannot be an instance of a type. Consider an enumeration type:

```
color = (red, green, blue)
```

Color is the type and red, green, and blue are its values. Most people would think of red, green and blue as instances of color. For BETA we ended up concluding that it is more natural to consider red, green and blue as subpatterns of color. The instances of say green are then all green objects. In Smalltalk True and False are subclasses of Boolean, but they are also objects. Numbers in Smalltalk are, however, considered instances of the respective number classes. For BETA we considered numbers to be subpatterns and not instances. Here we are in agreement with Hoare [55] that a value, like four, is an abstraction over all collections of four objects.

Language support for a value concept was a constant obstacle in the design of BETA. The enter/exit-part of a pattern, the unification of assignment and method invocation to some extent support the representation of a value concept. For Nygaard this was not enough and he constantly returned to the subject. Value type became an example of a concept that is well motivated from a modeling perspective, but it turned out to difficult to invent language mechanisms that added something new from a technical point of view.

5.8.3 Protection of attributes

There has been a lot of discussion of mechanisms for protecting the representation of objects. As mentioned, the introduction of abstract data types (where a data type was defined by means of its operations) and Hoare's paper on using the SIMULA class construct led to the introduction of **private** and **protected** constructs in SIMULA. Variants of **private** and **protected** are still the most common mechanism used in mainstream object-oriented languages like C++, Java and C#. In Smalltalk the rule is that all variables are private and all methods are public.

We found the private/protected constructs too ad hoc and the Smalltalk approach too restricted. Several proposals for BETA were discussed at that time, but none was found to be adequate.

5.8.4 Modularization

The concept of interface modules and implementation modules as found in Modula was considered a candidate for modularization in BETA. From a modeling point of view we needed a mechanism that would make it possible to separate the representative parts of a program – i.e. the part that represented phenomena and concepts from the application domain – from the pure implementation details. Interface modules and implementation modules were steps in the right direction.

However, we found that we needed more than just procedure signatures in interface modules, and we also found the concept of interface and implementation modules in conflict with the ‘one-pattern concept’. In our view,

modules were a mechanism that was used for two purposes: modularizing the program text and as objects encapsulating declarations of types, variables and procedures. In Section 5.8.8 we describe how the object aspect of a module may be interpreted as a BETA object.

For modularization of the program text we designed a mechanism based on the BETA grammar. In principle any sentential form – a correct sequence of terminal and nonterminal symbols from the BETA grammar – can be a module. This led to the definition of the fragment system, which is used for modularization of BETA programs. This includes separation of interface and implementation parts and separation of machine-dependent and independent parts. For details of the fragment system, see the BETA book [119].

5.8.5 Local language restriction

From the beginning of the project it was assumed that a pattern should be able to define a so-called local language restriction part. The idea was that it should be possible to restrict the use of a pattern and/or restrict the constructs that might be used in subpatterns of the pattern. This should be used when defining special purpose patterns for supporting class, procedure, function, type, etc. For subpatterns of e.g. a function pattern the use of global mutable data items and assignment should be excluded. Local language restriction was, however, never implemented as part of BETA, but remained a constant issue for discussion.

A number of special-purpose patterns were, however, introduced for defining external interfaces. These patterns are defined in an ad hoc manner, which may indicate that the idea of local language restriction should perhaps have been given higher priority.

5.8.6 Exception handling

Exception handling was not an issue when the BETA project started, but later it was an issue we had to consider. We did not like the dynamic approach to exception handling pioneered by Goodenough [45] and also criticized by Hoare [59]. As an alternative we adapted the notion of *static exception handling* as developed by Jørgen Lindskov Knudsen [72]. Knudsen has showed how virtual patterns may be used to support many aspects of exception handling and this style is being used in the Mjølner libraries and frameworks. The BETA static approach proved effective for exception handling in almost all cases, including large frameworks, runtime faults, etc. However, Knudsen [75] later concluded that there are cases (mostly related to third-party software) where static exception handling is not sufficient. In these cases there is a need either to have the compiler check the exception handling rules (as in e.g. CLU) or to introduce a dynamic exception handling concept in addition to the static one. In his paper he describes such a design and illustrates the strengths of combining both static and dynamic exception handling.

5.8.7 Ensemble

The relationship between the execution platform (hardware, and operating system) and user programs has been a major issue during the BETA project. As BETA was intended for systems programming, it was essential to be able to control the resources of the underlying platform such as processors, memory and external devices. An important issue was to be able to write schedulers.

The concept of ensemble was discussed for several years, and Dag Belsnes was an essential member of the team during that period. Various aspects of the work on ensembles have been described by the BETA team [93], Dag Belsnes [11], the BETA team [99], and Nygaard [131]. The first account of BETA's ensemble concept is in the thesis of Øystein Haugen [49].

A metaphor in the form of a theatre ensemble was developed to provide a conceptual/modeling understanding of an execution platform. A platform is viewed as an *ensemble* that is able to *perform* (execute) a *play* (program) giving rise to a *performance* (program execution). The ensemble has a set of *requisites* (resources) available in order to perform the play. Among the resources are a set of *actors* (processors). An actor is able to perform one or more *roles* (execute one or more objects) in the play. The casting of roles between actors (scheduling) is handled by the ensemble.

The interface to a given execution platform is described in terms of a BETA program including objects representing the resources of the platform. If a given platform has say four processors, the corresponding BETA program has four active objects representing the processors.

In addition to developing a conceptual understanding of an execution platform, the intention was to develop new language constructs. We think that we succeeded with the notion of ensemble as a concept. With respect to language constructs many proposals were made, but none of these turned out to be useful by adding new technical possibilities to the language. It turned out that the notions of active object and coroutine were sufficient to support the interface to processors and scheduling.

The ensemble concept did have some influence on the language. The Mjølner System includes an ensemble framework defining the interface to the execution platform. For most BETA implementations, one active object is representing the processor. A framework defines a basic scheduler, but users may easily define their own schedulers. An experimental implementation was made for a SPARC multiprocessor – here an active object was associated with each processor and a joint scheduler using these processors was defined as a framework.

Dynamic exchange of BETA systems. It was also a goal to be able to write a BETA program that could load and

execute other BETA programs. In an unpublished working note [99], we described a mechanism for '*Dynamic exchange of BETA systems*', which in some way corresponds to class loading in Java. Bjorn Freeman-Benson, Ole Agesen and Svend Frølund later implemented dynamic loaders for BETA.

Memory management was another issue we would have liked to support at the BETA level, but we did not manage to come up with a satisfactory solution.

5.8.8 Modules as objects

In the seventies the use of the class construct as a basis for defining abstract data types was often criticized since it implied an asymmetry between arguments of certain operations on a data type. Consider the following definition of a complex number:

```
class Complex:  
  { real x,y;  
    complex add(complex C) { ... }  
    ...  
  }  
  
Complex A,B,C;  
A := B.add(C);
```

The asymmetry in the call `B.add(C)` between the arguments `B` and `C` was considered by many a disadvantage of using classes to define abstract data types. As an alternative a module-like concept was proposed by Koster [79]:

```
module ComplexDef: {  
  type Complex = record real x,y end  
  Complex add(Complex C1,C2) { ... }  
  ...  
}  
  
Complex A,B,C;  
A := add(B,C);
```

As can be seen, this allows symmetric treatment of the arguments of `add`.

Depending on the language it was sometimes necessary to qualify the types and operation with the name of the module as in

```
ComplexDef.Complex A,B,C;  
A := ComplexDef.add(A,B);
```

Languages like Ada, CLU and Modula are examples of languages that used a module concept for defining abstract data types.

For BETA, a module was subsumed by the notion of *singular* object. The reason for this was that a module cannot be instantiated – there is only one instance of a module and its local types and operations can be instantiated. A complex module may be defined in BETA as follows:

```

ComplexDef: @
  (# Complex:
    (# X,Y: @real enter(X,Y) exit
     (X,Y#))
   add: (# ... #);
   ...
  #)

A,B,C: @ComplexDef.Complex;
(A,B) -> ComplexDef.add -> (B,C)

```

For CLU it was not clear to us whether the cluster concept was an abstraction or an object.

5.8.9 Constructors

The concept of constructors was often discussed in the project, but unfortunately a constructor mechanism was never included in BETA.

The idea of constructors for data types in general was introduced by Hoare (the idea was mentioned on page 55 top in [52] and the word constructor appears in [55]) and was obviously a good idea since it assured proper initialization of the objects. In SIMULA initialization of objects was handled by the do part of the object. As mentioned, all SIMULA objects are routines – when an object is generated it is immediately attached to the generating object and will thus start to execute its do part until it suspends execution. The convention in SIMULA was that initialization was provided by the code up to the first detach.

The SIMULA mechanism was not considered usable in BETA. In BETA an object is not necessarily a routine as in SIMULA. For BETA we wanted to support the notion of static procedure instance. This is illustrated by the example in Figure 8. The instance ia may be considered a static procedure instance and executed several times. We thought that it would not be meaningful to execute ia when it is generated. The do part of insert describes whatever insert should do and not its initialization.

We did consider having constructors in the style of C++, but we did not really like the idea of defining the constructor at the same level as the instance attributes (data items and procedures). We found constructors to be of the same kind as static procedures and static data items. As discussed elsewhere, we found static attributes superfluous in a block-structured language.

We liked the Smalltalk idea of a class object defining attributes like new to be global for a given class. Again, as described elsewhere, this should be expressed by means of block structure.

Unfortunately, the issue of constructors ended up as an example in which the search for a perfect solution ended up blocking a good solution – like C++ constructors.

5.8.10 Static (class) variables and methods

Static variables and methods were never an issue. In a block-structured language variables and methods global to a class naturally belong to an enclosing object. Static variables and methods play the roles of class variables and class methods in Smalltalk, and Madsen's paper on block structure [111] discusses how to model metaclasses and thereby class variables and class methods by means of block structure.

A further benefit of block structure is that one may have as many objects of the enclosing class as required (representing different sets of objects of the nested class), while static variables give rise to only one variable for all objects.

5.8.11 Abstract classes and interfaces

One implication of the one-pattern idea was that it was never an issue whether or not to have explicit support for abstract classes (or interfaces as found in Java). An abstract class was considered an abstraction mechanism on the line with class, procedure, type, etc.

If abstract class was to be included in BETA it would be similar to a possible support for class and procedure defined as patterns. I.e. one might imagine that BETA could have support for defining a pattern `AbstractClass` (or `Interface`).

For class and procedure we never really felt a need for defining special patterns. Since a pattern with only local patterns containing just their signature may be considered an abstract pattern, there was never a motivation to have explicit syntactic support for abstract patterns.

5.8.12 Multiple inheritance

BETA does not have multiple inheritance. In fact we did not like to use the term ‘inheritance’, but rather used ‘specialization’. This was deliberate: specialization is a relationship between a general pattern (representing a general concept) and patterns representing more special concepts, and with our conceptual framework as background this was most appealing. The specialized patterns should then have all properties of the more general pattern, and virtual properties could only be extended, not redefined. Inheritance should rather be a relationship between objects, as in everyday language. Specialized real-world phenomena cannot of course in general be substituted in the sense that they behave identically. But specialization implies substitutability in the following sense: a description (including program pieces) assuming certain properties of a general class of phenomena (like vehicles) should be valid no matter what kind of specialization (like car, bus or truck) of vehicle is substituted in the description (program piece). The description (program code) is safe in the sense that all properties are available but typically differ.

During the BETA project there was an ongoing discussion on multiple inheritance within object-oriented programming. Although one may easily recognize the need for multiple classifications of phenomena, the multiple inheritance mechanisms of existing languages were often justified from a pure code-reuse point of view: it was possible to inherit some of the properties of the superclasses but not all. Often there was no conceptual relation between a class and its multiple superclasses.

Language mechanisms for handling name conflicts between properties inherited from multiple superpatterns were a subject that created much interest. In one kind of approach they were handled by letting the order of superclasses define the visibility of conflicting names. From a modeling point of view it does not make sense for the order of the superclasses to be significant. In other approaches name conflicts should be resolved in the program text by qualifying an ambiguous name by the name of the superclass where it was declared. Name conflicts in general and as related to BETA were discussed by Jørgen Lindskov Knudsen [73]. He showed that no unifying name resolution rule can be devised since name conflicts can originate from different conceptual structures. The paper shows that there are essentially three necessary name-resolution rules and that these can coexist in one language, giving rise to great expressive power.

For BETA we would in addition have the complexity implied by inner, e.g. in which sequence should the superpattern actions be executed? There was a proposal that the order of execution should be nondeterministic. Kristine Thomsen [150] elaborated and generalized these ideas.

The heavy use of multiple inheritance for code sharing, and the lack of a need for multiple inheritance in real-world examples implied that we did not think that there was a strong requirement for supporting multiple inheritance. This was perhaps too extreme, but in order to include multiple inheritance the technical as well as modeling problems should be solved in a satisfactory way. We did not feel that we were able to do that.

In practice, many of the examples of multiple inheritance may be implemented using the technique with part objects as described in Section 5.5.1.

5.8.13 Mixins and method combination

In the beginning mixins, as known from Flavors [23], were never really considered for inclusion in BETA – i.e. covered by the pattern concept. The reason was that we considered mixins to be associated with multiple inheritance, and the concept of mixins seemed to be even further promoting multiple inheritance as a mechanism for code sharing. The semantics of multiple inheritance in Flavors, Loops [13] and Common Lisp [69] where the order of the superclasses was essential did not seem to fit

well with a language intended for modeling. Perhaps the emphasis on code sharing in these Lisp-based languages did not make us realize that a mixin can be used to define an aspect of a concept, as discussed in Section 5.5.1.

We found the support for method combination in these languages interesting. Before and after methods are an alternative – and perhaps more general – to the inner mechanism. Method combination is an interesting and important issue. Thomsen proposed a generalization of inner for combination of concurrent actions [151]. Bracha and Cook proposed a mixin concept supporting super as well as inner [15].

5.9 Syntax

It is often claimed that BETA syntax is awkward. It is noteworthy that these claims most often come from people not using BETA. Students attending BETA courses and programmers using BETA readily got used to it and appreciated its consistency. We could of course say that ‘syntax was not a big issue for us’ and ‘the semantics is the important issue’, but the fact is we had many discussions on syntax and that there is a reason why the syntax became the way it is.

First of all, we had the idea of starting with the desired properties of program executions and then making syntax that managed to express these properties. The terms object and object descriptor are simple examples of this.

Assignment: As we generalized assignment and procedure call into execution of objects, and as it was desired to have sequences of object executions, there was obviously a need to have a syntax that reflected what really was going on. The general form therefore became

```
ex1 -> ex2 -> ... -> exn
```

where each of the ex_i is an object execution. Execution involved assignment to the enter part, execution of the do part and assignment from the exit part.

Because references to objects could either be assigned to other references or be used in order to have the denoted object executed, we made the distinction syntactically:

```
ref1[] -> ref2[] ->...-> refn[]
```

```
(* reference assignment *)
```

```
ref1 -> ref2 ->...-> refn
```

```
(* object executions *)
```

The two forms could of course be mixed, so if the enter part of the object denoted by ref₂ required a reference as input, then that would be expressed by

```
ref1[] -> ref2
```

Naming: We devised the following consistent syntax for naming things and telling what they were:

```
<name> `:' <kind> <object descriptor>
```

By <kind> is meant *pattern*, *part object*, *reference variable*, etc. For part object the symbol @ is used to specify the kind and for reference variable ^ is used. For a pattern it was decided to use no symbol. So

```
P: super(# ... #)
```

simply was the syntax for a pattern. A possible super pattern was indicated by a name preceding the main part of the object descriptor and not (as in SIMULA) preceding the pattern name. Objects had two kinds and therefore different syntax: @ for a part object and ^ for a reference (to a separate object):

```
P:(# anA: @A;
  aRefToA: ^A;
  ...
  #)
```

Whenever a descriptor was needed (in order to tell e.g. what the properties of part object are) we allowed either an identifier (of a pattern) or a whole object descriptor:

```
P:(#
  anA1: @A;
  (* pattern-defined part object *)

  anA2: @(# ...#);
  (* singular part object *)

  aSpecialA: @A(# ...#);
  ...
  #)
```

The last part object above has an object descriptor that specializes A. This was made possible by the above syntax where a super pattern is indicated by a name preceding the main part of the object descriptor.

Parentheses: There are two reasons for using (# ... #) instead of { ... }. The first was that we imagined that there would be more than object descriptors that needed parentheses. At one point in time there were discussions about (@ ... @) meaning description of part objects and (= ... =) meaning description of values. This was never introduced, but later we introduced () to mean begin end of more than just object descriptors, e.g.

```
(for ... repeat ... for)
(if ... if)
```

We felt that this was obviously nicer than e.g.

```
for ... repeat ... endfor
if ... endif
```

found in other languages at that time. Although we did not introduce e.g. (@ ... @), we still reserved the # to mean descriptor, so that (# ... #) could be read ‘begin descriptor ... descriptor end’. The syntax for pattern variables uses # in order to announce that these are variables denoting descriptors.

5.10 Language evolution

In this section we briefly comment on how the language has evolved since 1976. Some of the events discussed below are also mentioned in Section 3.3.

The first account of BETA was the 1976 working note (First language draft [89]). At this stage the BETA project had mainly been concerned with discussing general concepts and sketching language ideas. A large part of the working note was devoted to a discussion of the DELTA concepts and their relation to BETA. The language itself was quite immature, but a first proposal for a pattern mechanism was presented. The report did not contain any complete program examples – an indication of the very early stage of the language.

The report includes a long analytical discussion of issues related to concurrency – this includes representative states and an interrupt concept. We had very little experience in issues related to concurrent programming. Various generalizations of the SIMULA coroutine mechanism were discussed. A lot of stacks were drawn and there were primitives like ATTACH X TO Y that could be used to combine arbitrary stacks. A few other language constructs were sketched, but not in an operational form – they were abandoned in future versions of BETA.

The syntax was quite verbose due to a heavy use of keywords. Parameters were passed by name and not by position. Objects had general enter/exit lists. The parameter mechanism made it possible to pass parameters and get return values to/from coroutines – something that is not possible in SIMULA.

The unification of name and procedure parameters and virtual procedures was mentioned but not described in the 1976 report. Virtual patterns were mentioned, and it was said that they would be as in SIMULA/DELTA).

The 1978 working note (Draft Proposal of BETA [90]) included a complete syntax, and the contour of the language started to emerge. Virtual patterns were used for method patterns, for typing functions and for typing elements of local arrays, that is virtual classes were in fact there. The syntax was very verbose with keywords, and very different from the final syntax, and the examples were sketchy.

The 1979 working note (First complete language definition [92]) included a complete definition of the language based on attribute grammars. In addition there were several examples.

With respect to language concepts, the 1981 working note (A survey of the BETA Programming Language) was quite similar to the 1979 working note, but there were major changes to the syntax. Most keywords were changed to special symbols: begin and end were replaced by (#, and

#); virtual and bind were replaced by :< and ::<; if and endif were replaced by (if and if); etc.

As mentioned previously, the POPL'83 paper on BETA (POPL: Abstraction Mechanisms [95]) was an important milestone. The POPL paper described the abstraction mechanisms of BETA. All the basic elements of BETA were in place including pattern, subpattern, block structure, virtual patterns and enter/exit. The syntax was almost as in the final version. The main difference was the use of a **pattern** keyword and different assignment operators like => and @> corresponding to := and :- in SIMULA. It was stated that the application area of BETA was embedded and distributed systems. The distinction between *basic* BETA and *standard* BETA with an extension of basic BETA with special syntax for a number of commonly used patterns was also stated. The POPL'83 paper contains a proposal for a generalization of the virtual pattern concept. The idea was that any syntactic category of the BETA grammar could be used as a virtual definition. The idea of generalized virtuals was, however, never further explored.

The POPL paper was accompanied with a paper describing the dynamic parts – coroutines, concurrency and communication. Communication and synchronization was based on CSP- and Ada-like rendezvous. We never managed to get the concurrency paper accepted at an international conference although we made several attempts – eventually the paper was published in Sigplan Notices [97] in 1985.

A combined version of the POPL paper and the concurrency paper was later (1987) included in the book that was published as a result of the Hawthorne workshop in 1986 [145]. However, the syntax was revised to that used in the final version of BETA.

Syntax Directed Program Modularization. A paper on syntax-directed program modularization was published at a conference in 1983 in Stresa [94] describing a proposal to program modularization based on the grammar (cf. Section 5.8.4). These principles for program modularization were further developed in the Mjølner project.

In the March 1986 revision of Dynamic Exchange of BETA Systems, the syntax was still not the final one although it differs from that of the POPL 83 paper.

From late 1986/early 1987, the sequential parts of BETA were stable in the sense that only a few changes were made. Pattern variables were added, the if statement was made deterministic, an else clause was added, and a few other minor details were changed.

During the Mjølner project, the rendezvous mechanism was replaced by the semaphore as a basic primitive for synchronization. In 1975 Jean Vaucher [157] had already shown how inner combined with prefixed procedures can

be used to define a monitor abstraction. This was immediately possible in BETA too. It also turned out that the pattern is well suited to build other higher-level concurrency abstractions, including Ada-like rendezvous and futures.

Many of the later papers on BETA were elaborations of the implications of the one-pattern approach. The simplicity of the pattern mechanism makes BETA simple to present, but the implications turned out to be difficult to convey. In many of the papers we therefore decided to use a keyword-based syntax and not the compact BETA syntax. Often redundant keywords like **class** and **proc** were introduced to distinguish between patterns used as classes and procedures. Some of the most important papers are the following:

- Classification of Actions – or Inheritance Also for Methods, presented at ECOOP'87 [101] and described how to use patterns and inner to define a hierarchy of methods and processes.
- What Object-Oriented Programming May Be and What It Does Not Have to Be, presented at ECOOP'88 [116]. Here we for the first time gave our definition of object-oriented programming and compared it with other perspectives on programming.
- Virtual Classes – a Powerful mechanism in Object-oriented Programming, which was presented at OOPSLA'89 [117]. The idea of virtual patterns was presented in the POPL'83 paper [95], but here the implications were presented in greater detail.
- Strong Typing of Object-Oriented Programming Revisited, presented at OOPSLA 90. The goal of this paper was to argue for our choice of covariance at the expense of runtime type checks.

The 1993 book on BETA [119] is the most comprehensive description of the language and the associated conceptual framework.

6. Implementations of BETA

During the first period of the BETA project, no attempts were made to implement a compiler. The reasons for this were mainly lack of resources: The implementation of SIMULA had been a major effort requiring a lot of resources. A number of large companies were involved in funding the SIMULA implementations, and we had nothing like this.

The SIMULA compilers were implemented in low-level languages – one of the compilers was even written in machine code. Implementation of the garbage collector, especially, had been a major task. In the beginning of the BETA project, we assumed that we would have to find funding for implementing BETA. We were thus working

from the assumption that we would have to establish a consortium of interested organizations.

There were other reasons than lack of funding. Nygaard was not a compiler person, Møller-Pedersen was employed by the NCC and could only use a limited amount of his time on BETA, and Kristensen and Madsen had to qualify for tenure.

In the early eighties an attempt was made to implement BETA by transforming a BETA program into a SIMULA program. The rationale for this was that we could then use the SIMULA compilers and run-time system for BETA. This project never succeeded – 90% of BETA was easy to map into SIMULA, but certain parts turned out to be too complicated.

During the BETA project, however, Kristensen and Madsen did substantial research on compiler technology and after some years realized that we had perhaps overestimated the job of implementing BETA.

6.1 The first implementation

The first implementation. In 1983 Madsen implemented the first BETA compiler in SIMULA. The first version generated code to an interpreter written in Pascal. The second version generated machine code for a DEC-10.

SUN Compiler. In 1985 this compiler was ported to a SUN workstation based on the Motorola 68020 microprocessor. This was an interesting exercise in bootstrapping. The SUN compiler was implemented using the DEC compiler, i.e. machine code was generated on the DEC-10 and transferred to the SUN for debugging. Since the turnaround time for each iteration was long, we manually corrected errors directly in the machine code on the SUN in order to catch as many errors as possible in each iteration. Afterwards such errors were fixed in the compiler. It was quite time-consuming and complicated to debug such machine code on the SUN.

The final step was to bootstrap the compiler itself. The DEC-10 was a slow machine and the BETA compiler was not very efficient. Using the compiler to compile itself was therefore a slow process. In addition, the DEC-10 was becoming more and more unstable – and it was decided that it should be closed down. The DEC-10 would not stay running for a whole compilation of the compiler. This meant that it was necessary to dump the state of the compiler at various stages and be able to restart it from such a dump in order to complete a full compilation of the compiler.

This was a complicated and time-consuming process and at one point Madsen did not believe that he would succeed. However, after three attempts, the bootstrapping succeeded and from then on the compiler was running on the SUN. This was a great experience.

The compiler implemented most parts of BETA – however, a garbage collector was not included. At that time we did not think that we had the qualifications to implement a garbage collector – we really needed some of the experienced people from the NCC.

6.2 The Mjølner implementations

The Mjølner Project provided the necessary time and resources to implement efficient working compilers for BETA. In the project it was decided to use the existing BETA compiler to implement the new compilers. Without a garbage collector this was not easy – a simple memory management scheme was added such that it was possible to mark the heap and subsequently release the heap until that mark. This was of course pretty unsafe, but we managed to implement the new compilers and the first versions of the MjølnerTool.

Knut Barra from the NCC wrote the first garbage collector [10] for BETA (as part of the SCALA project) and in 1987 a full workable implementation of BETA was available.

Macintosh Compiler. In the beginning of the Mjølner Project the SUN compiler was ported to a Macintosh. The Macintosh compiler was a special event. When we started working with Nygaard he did not use computers and he had not done any programming since the early sixties. When the Macintosh arrived we talked him getting a Mac and he quickly became a super user. It was therefore a great pleasure for us to be able to deliver the first Mac compiler to him on his 60th birthday in 1986.

Later in the Mjølner Project the compiler was ported to a number of machines, including Apollo and HP workstations. Nokia was a partner in the Mjølner Project. We ported BETA to an Intel-based telephone switch and implemented a remote debugger for BETA programs running on the switch on an Apollo workstation. This was a major improvement compared to the very long development cycles that were used by NOKIA for developing software for the switch.

The NOKIA compiler was later used as a basis for porting BETA to Intel-based computers running Windows or Linux. It took some time before these compilers were available since for many years the memory management on the Intel processors was based on segments, which were difficult to handle for a language with general references.

As of today there are or have been native BETA compilers for SUN, Apollo, HP, SGI, Windows, Linux and Macintosh.

6.3 The JVM, CLR, and Smalltalk VM compilers

In 2003 Peter Andersen and Madsen engaged in a project on language interoperability inspired by Microsoft .NET/CLR, which was announced as a platform supporting

language interoperability – in contrast to the Java/JVM platform. The goal of the project was to pursue to what extent CLR supported language interoperability. Another goal was to investigate to what extent this was supported by the JVM [9].

We managed to implement BETA on both JVM and CLR – i.e. full BETA compilers are running on top of JVM and CLR. The main difficulty was to make all the necessary type information from BETA available in the byte codes. Since BETA in many ways is more general than Java and C#, there are elements of BETA that do not map efficiently to these platforms. The most notorious example of this is coroutines, which are implemented on top of the thread mechanisms.

We are currently engaged in implementing BETA on a VM based on Smalltalk and intended for supporting pervasive computing – this VM is based on the Esmertec OSVM system and is being further developed in the PalCom project. One of the interesting features of this VM is that it has direct support for BETA-style coroutines.

6.4 Implementation aspects

We will touch only briefly on implementation aspects of BETA, since a complete description would take up a lot of space. The implementation is inspired by the SIMULA implementations [31, 122] and described by Madsen in the book about the Mjølner Project [112]. The generality of BETA implied that many people thought that it would be quite complicated (if not impossible) to make a reasonably efficient BETA implementation. Here are some of the major issues:

- **Virtual patterns.** The most difficult part of BETA to implement was virtual patterns. There are two aspects of virtual patterns: semantic analysis and run-time organization. The run-time organization was quite straightforward using a dispatch table. Semantic analysis appeared quite complicated – the problem was given the use of a virtual pattern to find the binding of the virtual that was visible at the point of use. The first attempt to write a semantic analyzer was made in a student project that failed, and for some time we were a bit pessimistic about whether or not we would succeed. It was not the virtual pattern concept by itself that was the real problem, but the combination with block structure. However, a (simple) solution was found and later documented by Madsen in a paper at OOPSLA'99: Semantic Analysis of Virtual Patterns [114].
- **Pattern.** The generality of the pattern concept imposed some immediate challenges for an efficient implementation. For a pattern (or singular object) used as a class, there should be code segments (routines) corresponding to generation (allocation and initialization of data items), enter, do and exit. For a pattern used as a

procedure there should just be one code segment. We originally assumed that the compiler could detect the use of a given pattern and generate code corresponding to the use. However, with separate compilation of pattern libraries, this is not possible. We ended up with a reasonable approach, but the code is not as efficient as it can be with separate constructs for class and procedure. In practice this has not been considered a problem. With modern just-in-time and adaptive compilation techniques, it should be straightforward to generate code for a pattern depending on its use.

▪ **Block-structure and subpatterns.** The relaxation of the SIMULA restriction that a subclass may be defined only at the same block level as its superclass gave rise to some discussion of whether or not this would have negative implications for an efficient implementation of block structure as described by Stein Krogdahl [106]. Since Algol, a variable in a block-structured language has been addressed by a pair, [block-level, offset]. By allowing subpatterns at arbitrary block levels, a variable is no longer identified by a unique block level: let x be declared in pattern P , let A and B be different subpatterns of P , and let A and B be at different block levels; then x in A is not at the same block level as x in B . We instead adapted the approach proposed by Wirth for Pascal to address a data-item by following the static link (*origin*) from the use of a data item to its declaration. This implied that an object has an *origin*-reference for each subclass that is not defined on the same block level as its superclass. For details see Madsen's implementation article [112].

▪ **The dynamic structure.** The implementation of the dynamic structure has been a subject for much discussion. Due to coroutines, SIMULA objects and activation-records are allocated on the heap. A similar scheme was adapted in the first BETA implementations, i.e. the machine stack was not used. Many people found this too inefficient and the implementation was later changed to use the machine stack. We do not know whether this makes a significant difference or not, since no systematic comparison of the two different techniques has been made. We do know that the heap-based implementation is significantly simpler than that using the machine stack. Whenever BETA has been ported to a new platform, stack handling has been the most time-consuming part to port. The generalization of inner implied that an object will need a *caller*-reference corresponding to each subclass with a non-empty do-part. For the heap-base implementations, these *caller*-references are stored in the object. For the stack-based implementations, the caller references are stored on the machine stack and thus not explicitly in the objects. We have also considered using the native stacks on modern operating systems, but these are too heavyweight for

coroutines – a program may allocate thousands of coroutines, which is beyond the capacity of these systems.

- **External interfaces.** No matter how nice, simple and safe a language you design, you will have to be able to interface to software implemented in other (unsafe) languages. For BETA a large number of external interfaces were made including C, COM, database systems, Java, and C#. This introduced major complications in the compiler since it was most often done on a by-need basis – often with little time for a proper design. In order to support various data types and parameters, BETA was polluted with patterns supporting e.g. pointers to simple data types like integers and C-structs. The handling of external calls further complicated the dynamic implementation since a coroutine stack may contain activations from external calls. If a callback is made from the external code, BETA activations may appear on top of the external stack. Perhaps the worst implication of this is that all BETA applications suffer from libraries and frameworks calling external code. The GUI-frameworks are examples of this: if they were used wrongly by the BETA programmer, the code was very difficult to debug. The lesson here is that external interfaces should be carefully designed and the implementation should encapsulate all external code in such a way that it cannot harm the BETA code – even though this may harm efficiency.

- **Garbage collection.** Over the years the Mjølner team became more and more experienced in writing garbage collectors and a number of different garbage collectors have been implemented varying from mark-sweep to generation-based scavenging. The first implementation of the Train algorithm was implemented for BETA by Jacob Seligmann and Steffen Grarup [144].

7. Impact

7.1 Teaching

BETA has been used for teaching object-oriented programming at a number of universities. The most important places we are aware of are as follows:

- **BETA courses in Aarhus.**

At DAIMI, BETA was an integral part of the curriculum at both the undergraduate and graduate level.

The Institute of Information Studies, Aarhus University is an interesting case, since this is a department in the Faculty of Humanities. Students within humanities traditionally have difficulties in learning programming. BETA was used for more than a decade and selected because of its clean and simple concepts, its modeling capabilities and its associated conceptual framework.

First draft of BETA book. A first draft of the BETA book [102] was made available (in the late eighties) to these students, and several versions of the BETA book [119] were tested here before the final version was printed. Originally all examples in the book were typical computer science examples such as stack, queue, etc. Such examples are not motivating for students within the humanities, and all the examples were changed to be about real world phenomena such as bank accounts, flight schedules, etc. Kim Halskov Madsen was very helpful in this process. Preprints of the BETA book were for many years distributed at OOPSLA and ECOOP by Mjølner Informatics and for many people these red books were their first encounter with BETA.

- **BETA courses in Oslo.** At the University of Oslo there were courses on specification of systems by means of SDL and BETA in 1988 and 1993 (by Møller-Pedersen and Dag Belsnes) and on object-oriented programming in BETA in 1994 and 1995 (by Møller-Pedersen, Nygaard and Ole Smørdal).
- **BETA courses in Aalborg.** At Department of Computer Science, University of Aalborg courses on object-oriented programming in BETA were given by Kristensen in 1995 and 1996.
- **BETA courses in Dortmund.** As mentioned, BETA was used for introductory programming at the University of Dortmund, Germany. Here the lecturers wrote a book in German on programming in BETA [38].

We believe that teaching of programming should be based on a programming language that reflects the current state of the art and is simple and general. Many schools use mainstream programming languages used in industry. Our experience is that it is easier to teach a state-of-the-art language than a standard industrial language. Students familiar with the state of the art can easily learn whatever industrial language they need to use in practice. The other way around is much more difficult. For BETA it was for many years necessary to argue that it was well suited for teaching. With the arrival of Java this changed, and Java took over at all places where BETA was used.

7.2 Research

In general BETA is well cited in the research literature. Perhaps the most influential part of BETA with respect to research is the concept of virtual class based on the use of virtual patterns as classes: Thorup [152], Bruce [19], Thorup [153], Mezini [126], and Odersky [134]. Other aspects of BETA such as inner, singular objects, block structure, and the pattern mechanism, have also been cited by many authors, e.g. Goldberg [44], and Igarashi and Pierce [62]. In 1994, Bill Joy designed a language without subclasses based on the ideas of inheritance from part objects as described in Section 5.5.1 [67]. Also in 1994, Bill Joy gave a talk in a SUN world-wide video conference

where he mentioned BETA as the most likely alternative to C++.

When we designed BETA we did not have deep enough knowledge of formal type theory to be able to establish the precise relations. In 1988/89 Madsen and others at DAIMI started discussions with Jens Palsberg and Michael Schwartzbach on applying type theory to object-oriented languages. Initially the hypothesis of Palsberg and Schwartzbach was that standard type theory could be applied, but they also realized that subtype substitutability and covariance were nontrivial challenges. This led to a series of papers on type theory and object-oriented languages [138] and a well-known book [140]. The main impact for BETA was that we learned that concepts like co- and contravariance were useful for characterizing virtual patterns in BETA. We had a hard time – and still have – relating to concepts such as universal and existential qualifiers, but more recent work has shed some light on this issue. Researchers with interests in such matters might think that virtual patterns are essentially existential types, but this view is too simplistic. One crucial difference, pointed out by Erik Ernst and explored in his work on family polymorphism [40], is that virtual classes rely on a simple kind of dependent types to allow more flexible usage: The unknown type, when bound by an existential quantifier, must be prevented from leaking out, whereas virtual classes can be used in a much larger scope, because the enclosing object can be used as a first-class package.

Schwartzbach and Madsen discussed making a complete formal specification of BETA's type system. Schwartzbach concluded at that time that the combination of block structure and virtual patterns made it very hard and we never succeeded. Igarashi and Pierce [61] and the authors mentioned below have over the years provided elements of formalization, including virtual classes and block structure.

Palsberg and Schwartzbach also did a lot of interesting work on type inference [139]. Two students of Schwartzbach implemented a system that could eliminate most (all) of the run-time checks in BETA and also detect the use of a given pattern and thereby optimize the code generation. The technique assumed a closed world, which made it less usable in a situation with precompiled libraries and frameworks. The work on type inference was later refined by Ole Agesen for Self [6, 7].

In 1997, Kresten Krab Thorup [152] published a paper on how to integrate virtual classes with Java. This was the starting point for a number of papers on virtual classes. In addition to Thorup, the work of Erik Ernst [39, 40], and Mads Torgersen [154] has been very decisive for interest in virtual classes. Several other researchers have elaborated on or been inspired by the virtual class concept, including work by Bruce, Odersky and Wadler [19] and Igarashi and Pierce [61]. Ernst has pointed out that some authors use the

term virtual type whereas he prefers (and we agree) the term virtual class. A virtual type may (only) be used to annotate variables whereas a virtual class may be used to create instances.

Erik Ernst has developed the language **gbeta**, which is a further generalization of the abstraction mechanisms of BETA. **gbeta** among others includes a type-safe dynamic inheritance mechanism [39]. **gbeta** also supports the use of virtual patterns as superpatterns. BETA did have a semantics for virtual patterns as superpatterns, and virtual super patterns were implemented in the first BETA compiler. They were, however, abandoned in later versions, since we never found a satisfactory efficient implementation. In **gbeta** the restrictions on virtual superpatterns are removed. In BETA it is possible to express a simple kind of dependent types by means of block-structure and virtual classes. This was identified and generalized by Ernst as the concept of family polymorphism [40]. The connection to existential types mentioned above builds on this notion of dependent types.

In order to have full static type checking, Torgersen has suggested forbidding invocation of methods with parameters that have a non-final virtual type [154]. For classes with such methods, a concrete subclass with all virtual types declared final must then be defined in order to invoke these methods.

The Scala language has *abstract type members*, which are closely related to virtual classes. Finally, the language Caesar [126] supports the notion of gbeta virtual classes in a Java context with some simplifications and restrictions.

At POPL'2006 [41], Erik Ernst, Klaus Ostermann, and William R. Cook presented a virtual class calculus that captures the essence of virtual classes. We think this is an important milestone because it is the first formal calculus with a type system and a soundness proof which directly and faithfully models virtual classes.

Ellen Agerbo and Aino Cornils [3] used virtual classes and part objects to describe some of the design patterns in The Gang of Four book [43].

In 1996, Søren Brandt and Jørgen Lindskov Knudsen made a proposal for generalizing the BETA type system [16]. The proposal generalizes the type system in two directions: first, by allowing type expressions that do not uniquely denote a class, but instead denote a closely related set of classes, and second, by allowing types that cannot be interpreted as predicates on classes, but must be more generally interpreted as predicates on objects. The resulting increase in expressive power serves to further narrow the gap between statically and dynamically typed languages, adding among other things more general generics, immutable references, and attributes with types not known until runtime.

Knudsen has made use of the BETA fragment system to support aspect-oriented programming [74].

Goldberg, Findler and Flatt [44] developed a language with both super and inner, arguing that programmers need both kinds of method combination. They also present a formal semantics for the new language, and they describe an implementation for MzScheme.

GOODS. Nygaard was the leader of General Object-Oriented Distributed Systems (GOODS), a three-year Norwegian Research Council-supported project starting in 1997. The aim of the project was to enrich object-oriented languages and system development methods by new basic concepts that make it possible to describe the relation between layered and/or distributed programs and the machine executing these programs. BETA was used as the foundation for the project and language mechanisms in BETA were studied, especially supporting the theatre ensemble metaphor. The GOODS team also included Haakon Bryhni, Dag Solberg and Ole Smørødal.

STAGE. The GOODS project continued in the STAGE Project (STAGing Environments) project at the NCC, aiming at establishing a commercial implementation of the GOODS idea. The STAGE team also included Dag Belsnes, Jon Skretting, and Kasper Østerbye. The project pursued the idea of the theater metaphor – cf. Section 5.8.7.

The Devise project. In 1990 three research groups at DAIMI decided to work together on research in tools, techniques, methods and theories for experimental system development. The groups were Coloured Petri Nets (headed by Kurt Jensen), systems work (HCI) (headed by Morten Kyng) and object-oriented programming (headed by Madsen). The rationale was that in order to make progress in system development, supplementary competences were needed. The implications for BETA were:

BETA was used as a common language for development of tools. One major example is the CPN Tool [28] for editing, simulating and analyzing Coloured Petri Nets. A unique characteristic of CPN Tools is that they were one of the first tools to use so-called post-WIMP interaction techniques, including tool glasses, marking menus, and bimanual interaction (using two mice). CPN Tools is in widespread use. Another major tool was a Dexter-based hypermedia [48], [47], [143]. A unique characteristic of this tool was the use of anchors that makes it possible to link between positions in different pages without modifying the pages. The hypermedia tool was the basis for a start-up company, Hypergenic Ltd.

BETA has played an important role in work on a multidisciplinary approach to experimental system development. Over the years the group developed techniques for people within programming, system development, participatory design, HCI and ethnography to

work together on software development projects, often using BETA and the Mjølner System. The object-oriented conceptual framework turned out to be a common framework and the graphical syntax of BETA supported by the Mjølner Tool turned out to be a useful means for communication between system developers and (expert) users [24].

From the beginning it was a goal to integrate Petri nets and object-oriented programming languages. The motivation was that in the early days of the BETA project Petri nets had a major influence on our conception of concurrency. Jensen, Kyng and Madsen started working together in formalizing DELTA using Petri nets. Jensen continued working with Petri nets and the group at DAIMI is well known internationally. Numerous suggestions for integrating object-orientation and Petri nets were investigated, but no real breakthrough was obtained. There are many suggestions in the literature for integrating Petri nets and object orientation, [108, 123].

The Devise group has continued to work together and now forms the basis of the Center for Pervasive Computing in Aarhus.

Conceptual Modeling and Programming. Design of programming languages could be based on human conceptualization in a more general sense. The approach was to include alternative kinds of concepts and selected ingredients of these concepts into programming languages in order to support modeling. The approach is described in [64, 104] and explored further in [124]. Object orientation could be seen as a specialized use of this approach, where the focus mainly is on “things” and their modeling in terms of classes and objects. The intention was that certain additional kinds of general (but not application area specific) concepts would enrich programming languages. The purpose was to limit the gap between understanding, designing and programming also in order to reduce the amount of software. The advantage of the approach is that because humans already use various alternative kinds of concepts, the modeling process is efficient and the model becomes understandable. The challenge was that any given potential kind of concept had to be understood and interpreted, and did not immediately comply with the typical understanding of programming languages. Each candidate concept should therefore be adjusted to fit with and slightly modify the expectations and possibilities at the programming level including implementation techniques. Candidate concepts include:

- Activities [81, 82, 103] are abstractions over collaborations of objects.
- Complex associations [83] are abstractions over complex relationships between structured objects.

- Roles [84, 105] are abstractions over the use of roles for objects as special relationships between objects.
- Relations [164, 166] are abstractions over relationships between objects.
- Subjective behavior [85] means abstraction over different views on objects from external and internal perspectives.
- Associations [86-88] are abstractions over collaboration, and include both structural and interaction aspects by integrating activity and role concepts.

7.3 Impact on language development

Object-oriented SDL. In 1986 Elektrisk Bureau (later ABB) asked Dag Belsnes and Møller-Pedersen to develop an object-oriented specification language. At the start the idea was to make this from scratch, but the project soon turned into an extension ([127], [128]) of the specification language SDL standardized by ITU – the International Telecommunication Union. BETA had an impact in the sense that concurrent processes of SDL became the candidate objects, in addition to the data objects that were also part of SDL. Users of SDL were primarily using processes, and as BETA had concurrent objects (and thereby patterns/subpatterns of these), it was obvious to do the same with SDL. The underlying model of SDL is that of a SDL system consisting of sets of nonsynchronized communicating processes, where the behavior of each process is described by a state machine. Introducing object orientation to this model implied the introduction of process types (in addition to sets of processes) and process subtypes defining specialization of state machine behavior. The inner concept was generalized to virtual transitions, i.e. transitions of a process type that may be redefined in process subtypes. In addition, the notion of virtual procedures was introduced, enabling parts of transitions to be redefined. In addition to constraints on virtual procedures, SDL also introduced default bindings. Virtual types (corresponding to virtual inner classes) were introduced, with constraints, both in terms of a supertype (as in BETA) and by means of a signature. In [21] it is demonstrated how this may be used to define frameworks; the same idea is pursued in [167]. Finally, types were extended with context parameters, a kind of generalized generic parameters, where also the constraints on the type parameters followed the BETA style of constraining. All of these extensions were standardized in the 1992 version of SDL [136].

Java. We do not claim that BETA had a major impact on Java, but as a curiosum we could mention that the two first commercial licenses of the Mjølner BETA System were acquired by James Gosling and Bill Joy.

Madsen was a visiting scientist at SUN Labs in 1994-95 when Java appeared on the scene – he was involved in

discussions on whether or not virtual types could be added to Java. However, this was never done.

Java includes final bindings and singular objects – called anonymous classes. Nested classes were later added to Java and called inner classes. As we understand, final bindings, anonymous classes and nested classes were inspired by BETA.

The recently added Wildcard mechanism [155] was developed by a research group at DAIMI based on research by Mads Torgersen, Kresten Krab Thorup, Erik Ernst and others and may be traced back to virtual patterns.

UML2.0. Shortly after Møller-Pedersen joined Ericsson in 1998, a number of UML users (including Ericsson) asked for a new and improved version of UML. On behalf of Ericsson Møller-Pedersen joined this work within OMG. The influence on UML2.0 was indirectly via SDL, i.e. the same kinds of concepts as in SDL were introduced in UML2.0 [50]. As an interesting observation, UML1.x had already classes with their own behavior, like in SIMULA and BETA, while (as mentioned above) most object programming languages do not have this. UML1.x also had nested classes, so the only new thing in UML2.0 is that they can be redefinable (i.e. virtual classes).

8. Conclusion

The BETA project has been an almost lifelong enterprise involving the authors, the late Kristen Nygaard and many other people. The approach to language design and informatics has been unusual compared to most other language projects we are aware of. The main reason is perhaps the emphasis on modeling, the working style, and the unusual organization of the project.

The project was supposed to be organized in a well-defined manner based on partners, contracts/grants and a firm working plan with milestones including a language specification in 1997. Since we did not succeed in obtaining this, the project continued for many years as an informal collaboration among the team members. If we had delivered a language specification in 1997 it would have been quite different from what BETA is today and probably less interesting. A project with firm deadlines and a firm budget might not have achieved the same result. Instead we were able to continue to invent, discuss, and reject ideas over many iterations. We could keep parts open where we did not have satisfactory solutions. It was never too late to come up with a complete new idea. We could continue to strive for the perfect language.

From 1986 when the Mjølner projects started, there was an organization around BETA – although Mjølner was not supposed to develop the BETA language. We had to finalize the language and make decisions for the parts that

were not complete and even make decisions we were not happy about.

The “one abstraction mechanism” idea was an important driving factor, but it may not have been unusual to base a language project on one or more initial ideas. In fact, one should never engage in language design without overall major ideas. Languages based on the current state of art may be well engineered but will not add to the state of the art. Such languages may be highly influential on praxis and we have seen many examples of that.

As time has passed, many new ideas for improving BETA have been proposed and new challenges have appeared. But for many years we found that most of the proposals would not make a real difference for the users of BETA. The work on updating the language, the documentation and software was simply not worth the effort. The time has, however, arrived for a new language in the SIMULA/BETA style, but the one or two real breaking ideas perhaps remain to be seen.

Nygaard’s system description (modeling) approach was an unusual approach to language design. Designing a programming language from a system description perspective is certainly different from basing it on whatever a computer can do or on a mathematical foundation.

Another unusual characteristic of the project was that we did not follow mainstream research in programming languages. As mentioned, Nygaard was not interested in the state of the art but left it to us. The advantage of this approach was that we were free to formulate new visions and not just focus on the next publication. Today most researchers seem mainly to focus on publishing minor improvements and solutions to state-of-the-art ideas. This does not create new big inventions.

The BETA project heavily influenced the participants and their relationships. We established lifelong valuable and appreciated personal and professional relationships. Being young and inexperienced researchers learning from working together with such an experienced person as Nygaard, many of our research attitudes were established during the project. The most valuable has been not to take established solutions for given, but rather question them, try to go for more general solutions, and to have alternative, ambitious, and long-reaching objectives.

Below we comment on the original research goals of the project.

One abstraction mechanism. We succeeded in developing the pattern as an abstraction mechanism subsuming most other abstraction mechanisms. Originally this was a theoretical challenge and we think that the pattern mechanism has proved its relevance and importance from a research perspective. The pattern mechanism has also

proved to be useful in teaching and practical programming. As a teaching tool it is beneficial to teach students the pattern mechanism as part of their first programming language, but probably only with success if the approach is supported strongly by the learning environment. Still, in order to appreciate the beauty of the pattern mechanism, the student has also to be familiar with the culture of the programming-language world including notions such as record, procedure, etc. Such cultural variations need to be appreciated before the unified, more abstract notion is relevant and appealing. For the skilled programmer who has already used several different programming languages, the presentation of the pattern mechanism seems to be a very fruitful experience. Such programmers typically learn yet another abstract level of programming and this knowledge is valuable through the daily life with the usual ordinary programming languages. Programmers with the opportunity to use the pattern for a longer period for real system development appreciate the freedom and powerfulness it supports.

The idea of one pattern replacing all other abstraction mechanisms worked out well in practice. The unification clearly implied a simplification of the language, just as the extra benefits as mentioned in Section 5.1.2 clearly paid off. We occasionally hear people complain that they find it to be a disadvantage that they cannot see from a pattern declaration whether it is a class or method.

Virtual patterns turned out to be a major strength of BETA – the use of virtual patterns as virtual classes/types has in addition provided the basis for further research by many others.

Singular objects, block structure, etc. have also proved their value in practice and are heavily used by all BETA programmers. These mechanisms are also starting to arrive in other languages.

The enter-exit mechanism is of course used for defining parameters and return values for methods – in addition, it is used for defining value types. Many people make heavy use of enter/exit for overloading assignment and/or reading the value of an object. Although the enter/exit-mechanism has turned out to be quite useful in practice, it does have some drawbacks. The name of an argument has to be declared twice – once with a type and then in the enter/exit-part – this is similar to Algol and SIMULA but is, however inconvenient for simple parameters. In addition, the implementation of enter/exit in its full generality turned out to be quite complex.

A constructor mechanism is perhaps the most profound language element that is missing in BETA.

Couroutines and concurrency. We think that BETA has further demonstrated the usefulness of the SIMULA coroutine mechanism to support concurrency and

scheduling. The coroutine mechanism together with the semaphore turned out to fulfill the original goals. The implementation was simple and straightforward, and it has showed its usefulness in practice.

In addition, the abstraction mechanisms of BETA have proved their usefulness in defining higher-order abstraction mechanisms. The BETA libraries contain several examples of high-level concurrency abstractions. Few people in practice, however, define their own concurrency abstractions. Most concurrency abstractions have been defined by the authors and implementers of BETA.

In general concurrency and the ability to define concurrency abstractions are not as heavily used as we think they should be. This may be due to the fact that concurrency has not been an integrated part of most object-oriented languages. Java has concurrency but as a fixed synchronization mechanism in the form of monitor – there are no means for defining other concurrency abstractions including schedulers. We think that it should be an integrated part of the design of frameworks and components also to define the associated concurrency abstractions including schedulers.

We also think that SIMULA/BETA style coroutines are yet to be discovered by other language designers.

Efficiency. The original goal of proving that an object-oriented programming language could be efficiently implemented turned out to be less important. Several successors to SIMULA starting with C++ proved this. In addition, a number of efficient implementation techniques and more efficient microprocessors have implied that lack of efficiency is hardly an issue anymore.

Modeling. The modeling approach succeeded in the sense that a comprehensive conceptual framework has been developed. The conceptual framework consists of a collection of conceptual means for understanding and organizing knowledge about the real world. It is furthermore described how these means are related to programming language constructs. But just as important, it is emphasized that some conceptual means are not supported by BETA and other programming languages. As mentioned previously, we think that it is necessary for software developers to be aware of a richer conceptual framework than that supported by a given language. Otherwise the programming language easily limits the ability of the programmer to understand the application domain. A conceptual framework that is richer than current programming languages can be used to define requirements for new programming languages. This leads to the other point where we think that the modeling approach has succeeded.

We have demonstrated that language constructs and indeed a whole language can be based on a modeling approach. As

we hope we have demonstrated in this paper, almost all constructs in BETA are motivated by their ability to model properties of the application domain. They also had to have properties from a technical point of view and to be sufficiently primitive in order to be efficiently implemented. The art of designing a programming language is to balance the support of conceptual means and selection of primitives that may be efficiently implemented. We did e.g. not include dynamic classification and equations since we did not find that we could implement such constructs efficiently.

The goal for BETA was to design a language that could be used for modeling as well as programming. For many years the programming language community was not interested in modeling, and when object-orientation started to become popular, the main focus was on extensibility and reuse of code. This changed when the methodology schools started to become interested in object-oriented analysis and design. The approach to modeling in these schools was, however, different from ours. Most work on modeling aimed at designing special modeling languages based on a graphical syntax. As mentioned in Section 4, this reintroduced some of the problems of code generation and reverse engineering known from SA/SD. For BETA it was important to stress that the same language can be used for modeling as well as for programming and that syntax is independent of this. This was stressed by the fact that we designed both a textual and a graphical syntax for BETA. The attempts in recent years to design executable modeling languages in our opinion emphasizes that it was not a good idea to have separate modeling and programming languages.

There is no doubt that object orientation has become the mainstream programming paradigm. There are hundreds (or thousands) of books introducing object-oriented programming and methodologies based on object orientation. The negative side of this is that the modeling aspect that originated with SIMULA seems to be disappearing. Very few schools and books are explicit about modeling. It is usually restricted to a few remarks in the introduction; the rest of the book is then concerned with technical aspects of a programming language or UML or traditional software methodology.

We think that some of the advantages of object orientation have disappeared in its success and that there might be a need for proper reintroduction of the original concepts. OOA and OOD are in most schools nothing more than just programming at a high level of abstraction corresponding to the application domain. In order to put more content into this, there is room for making more use of the parts of the conceptual framework of BETA that go beyond what is supported by current programming languages. This would improve the quality of the analysis and design phases. We also think that future languages should be designed for

modeling as well as programming. Turning a modeling language into a programming language (or vice versa) may not be the best approach.

9. Acknowledgments

The Joint Language Project included at least: Bjarne Svejgaard, Leif Nielsen, Erik Bugge, Morten Kyng, Benedict Løfstedt, Jens Ulrik Mouritsen, Peter Jensen, Nygaard, Kristensen, and Madsen.

In addition to Nygaard and the authors, a large number of people have been involved in the development of BETA and the Mjølner BETA System including colleagues and students at Aarhus University, The Norwegian Computing Center, Oslo University, Aalborg University, and Mjølner Informatics A/S. In particular, the contributions of Dag Belsnes and Jørgen Lindskov Knudsen are acknowledged.

Development of the Mjølner BETA System in the Nordic Mjølner Project has been supported by the Nordic Fund for Technology and Research. Participants in the Mjølner Project came from Lund University, Telesoft, EB Technology, The Norwegian Computing Center, Telenokia, Sysware ApS, Aalborg University and Aarhus University. The main implementers of the Mjølner BETA System were Peter Andersen, Lars Bak, Søren Brandt, Jørgen Lindskov Knudsen, Henry Michael Lassen, Ole Lehrmann Madsen, Kim Jensen Møller, Claus Nørgaard, and Elmer Sandvad. In addition, Peter von der Ahe, Knut Barra, Bjorn Freeman-Benson, Karen Borup, Michael Christensen, Steffen Grarup, Morten Grouleff, Mads Brøgger Enevoldsen, Søren Smidt Hansen, Morten Elmer Jørgensen, Kim Falck Jørgensen, Karsten Strandgaard Jørgensen, Stephan Korsholm, Manmathan Muthukumarapillai, Peter Ryberg Jensen, Jørgen Nørgaard, Claus H. Pedersen, Jacob Seligmann, Lennert Sloth, Tommy Thorn, Per Fack and Peter Ørbæk have participated in various parts of the implementation.

In addition to the people mentioned above, we have had the pleasure of being engaged in research in object-orientation including issues related to BETA with a large number of people including Boris Magnusson, Görel Hedin, Erik Ernst, Henrik Bærbak Christensen, Mads Torgersen, Kresten Krab Thorup, Kasper Østerbye, Aino Cornils, Kristine S. Thomsen, Christian Damm, Klaus Marius Hansen, Michael Thomsen, Jawahar Malhotra, Preben Mogensen, Kaj Grønbæk, Randy Trigg, Dave Ungar, Randy Smith, Urs Hözle, Mario Wolczko, Ole Agesen, Svend Frølund, Michael H. Olsen, and Alexandre Valente Sousa.

The writing of this paper has happened during a period of nearly two years. We would like to thank the HOPL-III Program Committee for their strong support and numerous suggestions. Our shepherds Julia Lawall and Doug Lea have been of great help and commented on everything

including content, structure and our English language style. The external reviewers, Andrew Black, Gilad Bracha, Ole Agesen, and Koen Classen, have also provided us with detailed and helpful comments. Finally, we have received suggestions and comments from Peter Andersen, Jørgen Lindskov Knudsen, Stein Krogdahl, and Kasper Østerbye. We are grateful to Susanne Brøndberg who helped correct our English and to Katrina Avery for the final copy editing.

The development of BETA has been supported by the Danish Research Council, The Royal Norwegian Council for Scientific and Industrial Research, The European Commission, The Nordic Fund for Technology and Research, Apple Computer, Apollo, Hewlett Packard, Sun Microsystems, Microsoft Denmark, and Microsoft Research Cambridge.

As said in Section 3, the BETA project have had a major influence on our personal life and our families have been deeply involved in the social interaction around the project. We thank from the Kristensen family: Lis, Unna, and Eline; from the Madsen family: Marianne, Christian, and Anne Sofie; from the Møller-Pedersen family: Kirsten, Kamilla, and Kristine; and Johanna Nygaard. When the project started Christian had just been born – by now we are all grandparents.

10. References

- [1] Webster's New World Compact School and Office Dictionary: Prentice Hall Press, 1982.
- [2] Ada: Ada Reference Manual. Proposed Standard Document: United States Department of Defense, 1980.
- [3] Agerbo, E. and Cornils, A.: *How to Preserve the Benefits of Design Patterns*, OOPSLA'98 – ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, Vancouver, British Columbia, Canada, 1998, Sigplan Notices ACM Press.
- [4] Agesen, O., Frølund, S., and Olsen, M. H.: Persistent and Shared Objects in BETA, Master thesis, Computer Science Department, Aarhus University, Aarhus 1989.
- [5] Agesen, O., Frølund, S., and Olsen, M. H.: *Persistent Object Concepts*, in *Object-Oriented Environments—The Mjølner Approach*, Knudsen, J. L., Löfgren, M., Madsen, O. L., and Magnusson, B., Eds.: Prentice Hall, 1994.
- [6] Agesen, O., Palsberg, J., and Schwartzbach, M. I.: *Type Inference of SELF*, ECOOP'93 – European Conference on Object-Oriented Programming, Kaiserslautern, 1993, Lecture Notes in Computer Science vol. 707, Springer.
- [7] Agesen, O. and Ungar, D.: *Sifting Out the Gold*, OOPSLA'94 – Object-Oriented Programming Systems, Languages and Applications, Portland, Oregon, 1994, Sigplan Notices vol. 29, ACM Press.
- [8] America, P.: *Inheritance and Subtyping in a Parallel Object-Oriented Language*, ECOOP'87 – European Conference on Object-Oriented Programming, 1987, Lecture Notes in Computer Science vol. 276, Springer Verlag.

- [9] Andersen, P. and Madsen, O. L.: *Implementing BETA on Java Virtual Machine and .NET—an Exercise in Language Interoperability*, unpublished manuscript, 2003.
- [10] Barra, K.: *Mark/sweep Compaction for Substantially Nested Beta Objects*, Norwegian Computing Center, Oslo, DTEK/03/88, 1988.
- [11] Belsnes, D.: *Description and Execution of Distributed Systems*, Norwegian Computing Center, Oslo, Report No 717, 1982.
- [12] Blake, E. and Cook, S.: *On Including Part Hierarchies in Object-Oriented Languages, with an Implementation in Smalltalk*, ECOOP'87 – European Conference on Object-Oriented Programming, Paris, 1987, Lecture Notes in Computer Science vol. 276, Springer Verlag.
- [13] Bobrow, D. G. and Stefik, M.: *The LOOPS Manual*, Palo Alto AC: Xerox Corporation, 1986.
- [14] Booch, G.: *Object-Oriented Analysis and Design with Applications*. Redwood City: Benjamin/Cummings, 1991.
- [15] Bracha, G. and Cook, W.: *Mixin-based Inheritance*, Joint OOPSLA/ECOOP'90 – Conference on Object-Oriented Programming: Systems, Languages, and Applications & European Conference on Object-Oriented Programming, Ottawa, Canada, 1990, Sigplan Notices vol. 25, ACM Press
- [16] Brandt, S. and Knudsen, J. L.: *Generalising the BETA Type System*, ECOOP'96 – Tenth European Conference on Object-Oriented Programming Linz , Austria, 1996, Springer Verlag.
- [17] Brinch-Hansen, P.: *The Programming Language Concurrent Pascal*, IEEE Transactions on Software Engineering, vol. SE-1(2), 1975.
- [18] Brinch-Hansen, P.: *The Origin of Concurrent Programming: From Semaphores to Remote Procedure Calls*: Springer, 2002.
- [19] Bruce, K., Odersky, M., and Wadler, P.: *A Statically Safe Alternative to Virtual Types*, ECOOP'98 – European Conference on Object-Oriented Programming, Brussels, 1998, Lecture Notes in Computer Science vol. 1445, Springer Verlag.
- [20] Bruce, K. and Vanderwaart, J. C.: *Semantics-Driven Language Design: Statically Type-safe Virtual Types in Object-Oriented Languages*, Fifteenth Conference on the Mathematical Foundations of Programming Semantics, 1998.
- [21] Bræk, R. and Møller-Pedersen, B.: *Frameworks by Means of Virtual Types—Exemplified by SDL*, IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XI) and Protocol Specification, Testing and Verification (PSTV XVIII), 1998.
- [22] Budd, T.: *An Introduction to Object-Oriented Programming, third edition*: Addison Wesley, 2002.
- [23] Cannon, H.: *Flavors: A Non-Hierarchical Approach to Object-Oriented Programming*, Symbolics Inc., 1982.
- [24] Christensen, M., Crabtree, A., Damm, C. H., Hansen, K. M., Madsen, O. L., Marqvardsen, P., Mogensen, P., Sandvad, E., Sloth, L., and Thomsen, M.: *The M.A.D Experience: Multi-Perspective Application Development in Evolutionary Prototyping*, ECOOP'98 – European Conference on Object-Oriented Programming, Brussels, 1998, Lecture Notes in Computer Science vol. 1445, Springer Verlag.
- [25] Coad, P. and Yourdon, E.: *Object-Oriented Analysis*. Englewood Cliffs, N.J: Prentice-Hall, 1991.
- [26] Cook, S.: *Impressions of ECOOP'88*, Journal of Object-Oriented Programming, vol. 1, 1988.
- [27] Cook, W.: *Peek Objects*, ECOOP'2006 – European Conference on Object-Oriented Programming, Nantes, France, 2006, Lecture Notes in Computer Science vol. 4067, Springer Verlag.
- [28] CPNTOOLS: *Computer Tool for Coloured Petri Nets*.
- [29] Dahl, O.-J., Dijkstra, E. W., and Hoare, C. A. R.: *Structured Programming*: Academic Press, 1972.
- [30] Dahl, O.-J. and Hoare, C. A. R.: *Hierarchical Program Structures*, in *Structured Programming*: Academic Press, 1972.
- [31] Dahl, O.-J. and Myhrhaug, B.: *SIMULA 67 Implementation Guide*, Norwegian Computing Center, Oslo, NCC Publ. No. S-9, 1969.
- [32] Dahl, O.-J., Myhrhaug, B., and Nygaard, K.: *SIMULA 67 Common Base Language (Editions 1968, 1970, 1972, 1984)*, Norwegian Computing Center, Oslo, 1968.
- [33] Dahl, O.-J. and Nygaard, K.: *SIMULA—a Language for Programming and Description of Discrete Event Systems*, Norwegian Computing Center, Oslo, 1965.
- [34] Dahl, O.-J. and Nygaard, K.: *SIMULA: an ALGOL-based Simulation Language*, Communications of the ACM, vol. 9, pp. 671–678, 1966.
- [35] Dahl, O.-J. and Nygaard, K.: *The Development of the SIMULA Languages*, ACM SIGPLAN History of Programming Languages Conference, 1978.
- [36] Dijkstra, E. W.: *Go To Considered Harmful*, Letter to Communications of the ACM, vol. 11 pp. 147–148, 1968.
- [37] Dijkstra, E. W.: *Guarded Commands, Nondeterminacy and the Formal Derivation of Programs*, Communications of the ACM, vol. 18, pp. 453–457, 1975.
- [38] Doberkat, E.-E. and Dißmann, S.: *Einführung in die objektorientierte Programmierung mit BETA*: Addison Wesley Longman Verlag GmbH, 1996.
- [39] Ernst, E.: *Dynamic Inheritance in a Statically Typed Language*, Nordic Journal of Computing, vol. 6, pp. 72–92, 1999.
- [40] Ernst, E.: *Family Polymorphism*, ECOOP'01 – European Conference on Object-Oriented Programming, Budapest, Hungary, 2001, Lecture Notes in Computer Science vol. 2072, Springer Verlag.
- [41] Ernst, E., Ostermann, K., and Cook, W. R.: *A Virtual Class Calculus*, The 33rd Annual ACM SIGPLAN – SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, 2006.
- [42] Folke Larsen, S.: *Egocentrisk tale, begrebsudvikling og semantisk udvikling*., Nordisk Psykologi, vol. 32(1), 1980.
- [43] Gamma, E., Helm, R., Johnson, R., and Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*: Addison-Wesley, 1995.

- [44] Goldberg, D. S., Findler, R. B., and Flatt, M.: *Super and Inner—Together at Last!*, OOPSLA'04 – Object-Oriented Programming, Systems Languages and Applications, Vancouver, British Columbia, Canada., 2004, SIGPLAN Notices vol. 39, ACM.
- [45] Goodenough, J. B.: *Exception Handling: Issues and a Proposed Notation*, Communications of the ACM, vol. 18, pp. 683–696, 1975.
- [46] Gosling, J., Joy, B., and Steele, G.: *The Java (TM) Language Specification*: Addison-Wesley, 1999.
- [47] Grønbæk, K. and Trigg, R. H.: *Design Issues for a Dexter-based Hypermedia System*, Communications of the ACM, vol. 37, pp. 40–49, 1994.
- [48] Halasz, F. and Schwartz, M.: *The Dexter Hypertext Reference Model*, Communications of the ACM, vol. 37, pp. 30–39, 1994.
- [49] Haugen, Ø.: Hierarkibegreber i Programmering og Systembeskrivelse (Hierarchy Concepts in Programming and System Description), Master thesis, Department of Informatics, University of Oslo, Oslo 1980.
- [50] Haugen, Ø., Møller-Pedersen, B., and Weigert, T.: *Structural Modeling with UML 2.0*, in *UML for Real*, Lavagno, L., Martin, G., and Selic, B., Eds.: Kluwer Academic Publishers, 2003.
- [51] Hejlsberg, A., Wiltamuth, S., and Golde, P.: *The C# Programming Language*: Addison-Wesley, 2003.
- [52] Hoare, C. A. R.: *Record Handling*, ALGOL Bulletin, 1965.
- [53] Hoare, C. A. R.: *Further Thoughts on Record Handling*, ALGOL Bulletin, 1966.
- [54] Hoare, C. A. R.: *Record Handling—Lecture Notes*, NATO Summer School September 1966, in *Programming Languages*, Genuys, Ed.: Academic Press 1968, 1966.
- [55] Hoare, C. A. R.: *Notes on Data Structuring*, in *Structured Programming*. London: Academic Press, 1972.
- [56] Hoare, C. A. R.: *Proof of Correctness of Data Representations*, Acta Informatica, vol. 1, 1972.
- [57] Hoare, C. A. R.: *Monitors: An Operating Systems Structuring Concept*, Communications of the ACM, 1975.
- [58] Hoare, C. A. R.: *Communicating Sequential Processes*, Communications of the ACM, vol. 21, 1978.
- [59] Hoare, C. A. R.: *Turing Award 1980 Lecture: "The Emperor's Old Clothes"*, Communications of the ACM, vol. 24, pp. 75–83, 1981.
- [60] Holbæk-Hanssen, E., Håndlykken, P., and Nygaard, K.: *System Description and the DELTA Language*, Norwegian Computing Center, Oslo, Report No 523, 1973.
- [61] Igarashi, A. and Pierce, B. C.: *Foundations for Virtual Types*, The Sixth International Workshop on Foundations of Object-Oriented Languages – FOOL 6, San Antonio, Texas, 1999.
- [62] Igarashi, A. and Pierce, B. C.: *On Inner classes*, Information and Computation, vol. 177, 2002. A special issue with papers from the 7th International Workshop on Foundations of Object-Oriented Languages (FOOL7). An earlier version appeared in Proceedings of ECOOP2000 – 14th European Conference on Object-Oriented Programming, Springer LNCS 1850, pages 129–153, June, 2000.
- [63] Jackson, M.: *System Development*: Prentice-Hall, 1983.
- [64] Jacobsen, E. E.: Concepts and Language Mechanisms in Software Modelling, PhD thesis, University of Southern Denmark, 2000.
- [65] Jensen, K., Kyng, M., and Madsen, O. L.: *A Petri Net Definition of a System Description Language*, in *Semantics of Concurrent Computations*, Evian, Kahn, G., Ed.: Springer Verlag, 1979.
- [66] Jensen, P. and Nygaard, K.: *The BETA Project*, Norwegian Computing Center, Oslo, Publication No 558, 1976.
- [67] Joy, B.: *Personal Communication*. Sun Microsystems, 1994.
- [68] Kay, A.: *Microelectronics and the Personal Computer*, Scientific America, vol. 237(3), pp. 230–244, 1977.
- [69] Keene, S. E.: *Object-Oriented Programming in COMMON Lisp—a Programmer's Guide to CLOS*: Reading MA: Addison Wesley 1989.
- [70] Knudsen, J. L.: *Implementing BETA Communication, a Proposal*, unpublished manuscript, 1981.
- [71] Knudsen, J. L.: Aspects of Programming Languages: Concepts, Models and Design, thesis, Department of Computer Science, University of Aarhus, Aarhus 1982.
- [72] Knudsen, J. L.: *Exception Handling—A Static Approach*, Software Practice and Experience, 1984.
- [73] Knudsen, J. L.: *Name Collision in Multiple Classification Hierarchies*, ECOOP'88 – 2nd European Conference on Object-Oriented Programming, Oslo, 1988, Lecture Notes in Computer Science vol. 322, Springer Verlag.
- [74] Knudsen, J. L.: *Aspect-Oriented Programming in BETA Using the Fragment System*, Workshop on Object-Oriented Technology, 1999, Lecture Notes In Computer Science vol. 1743.
- [75] Knudsen, J. L.: *Fault Tolerance and Exception Handling in BETA*, in *Advances in Exception Handling Techniques*, vol. 2022, Lecture Notes in Computer Science, Romanovsky, A., Dony, C., Knudsen, J. L., and Tripathi, A., Eds.: Springer Verlag, 2001.
- [76] Knudsen, J. L., Löfgren, M., Madsen, O. L., and Magnusson, B.: *Object-Oriented Environments—The Mjølner Approach*: Prentice Hall, 1993.
- [77] Knudsen, J. L. and Madsen, O. L.: *Teaching Object-Oriented Programming is more than Teaching Object-Oriented Programming Languages*, ECOOP'88 – European Conference on Object Oriented Programming, Oslo, 1988, Lecture Notes in Computer Science vol. 322, Springer Verlag.
- [78] Knudsen, J. L. and Thomsen, K. S.: A Conceptual Framework for Programming Languages, Master thesis, Computer Science Department, Aarhus University, Aarhus 1985.
- [79] Koster, C. H. A.: *Visibility and Types*, SIGPLAN 1976 Conference on Data, Salt Lake City, Utah, USA, 1976.
- [80] Kreutzer, W. and Østerbye, K.: *BetaSIM—a Framework for Discrete Event Modelling and Simulation*, Simulation Practice and Theory, vol. 6, pp. 573–599, 1998.

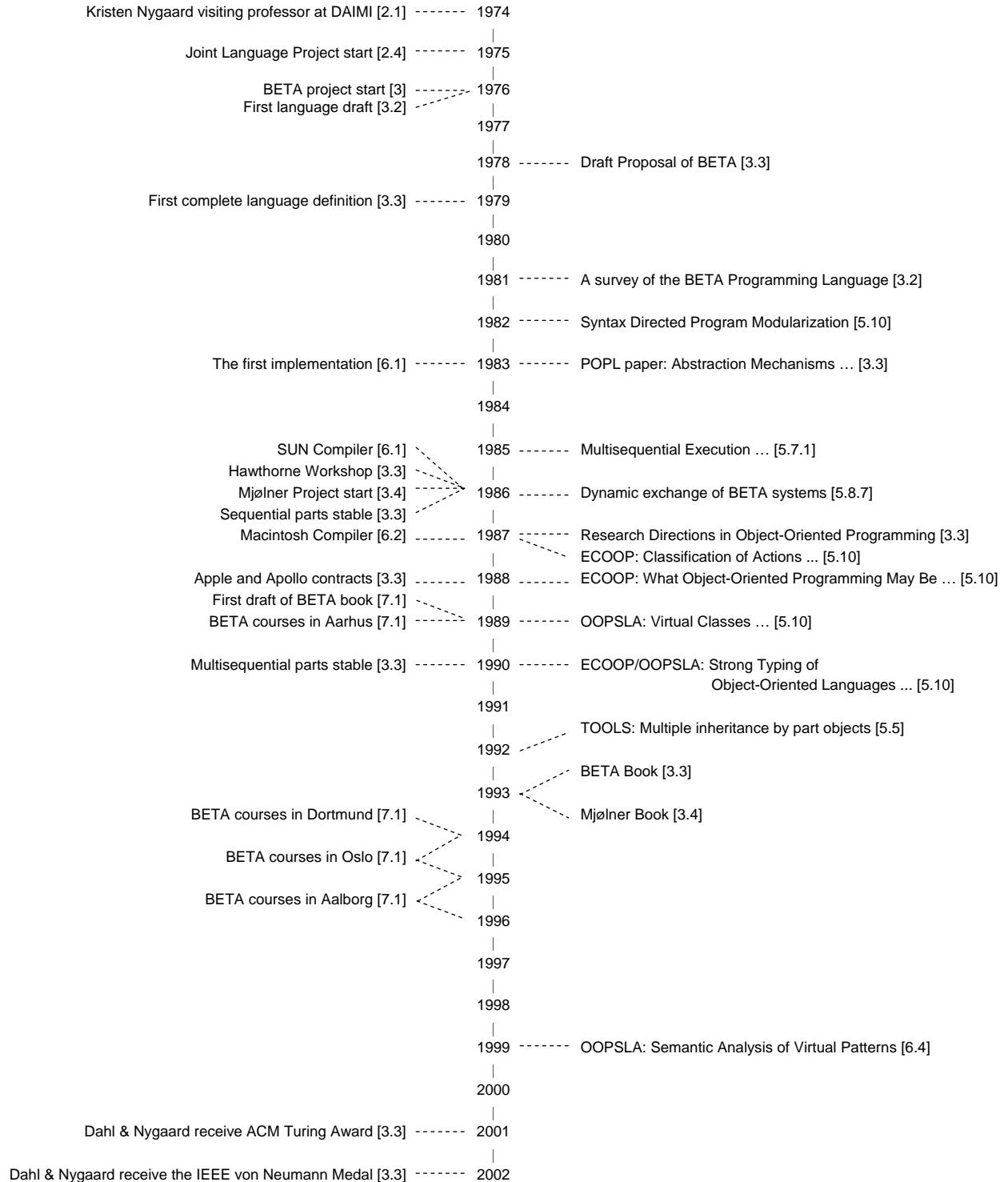
- [81] Kristensen, B. B.: *Transverse Activities: Abstractions in Object-Oriented Programming*, International Symposium on Object Technologies for Advanced Software (ISOTAS'93), Kanazawa, Japan, 1993.
- [82] Kristensen, B. B.: *Transverse Classes & Objects in Object-Oriented Analysis, Design and Implementation*, Journal of Object-Oriented Programming, 1993.
- [83] Kristensen, B. B.: *Complex Associations: Abstractions in Object-Oriented Modeling*, OOPSLA'94 – Object-Oriented Programming Systems, Languages and Applications, Portland, Oregon, 1994, Sigplan Notices vol. 29(10), ACM Press.
- [84] Kristensen, B. B.: *Object-Oriented Modeling with Roles*, 2nd International Conference on Object-Oriented Information Systems (OOIS'95), Dublin, Ireland, 1995.
- [85] Kristensen, B. B.: *Subjective Behavior*, International Journal of Computer Systems Science and Engineering, vol. 16, pp. 13–24, 2001.
- [86] Kristensen, B. B.: *Associative Modeling and Programming*, 8th International Conference on Object-Oriented Information Systems (OOIS'2002), Montpellier, France, 2002.
- [87] Kristensen, B. B.: *Associations: Abstractions over Collaboration*, IEEE International Conference on Systems, Man & Cybernetics, Washington D. C., 2003.
- [88] Kristensen, B. B.: *Associative Programming and Modeling: Abstractions over Collaboration*, 1st International Conference on Software and Data Technologies, Setúbal, Portugal, 2006.
- [89] Kristensen, B. B., Madsen, O. L., Møller-Pedersen, B., and Nygaard, K.: *BETA Language Development, Survey Report, 1. November 1976*, Norwegian Computing Center, Oslo, Report No 559, 1977.
- [90] Kristensen, B. B., Madsen, O. L., Møller-Pedersen, B., and Nygaard, K.: *DRAFT PROPSAL for Introduction to The BETA Programming Language as of 1st August 1978*, Norwegian Computing Center, Oslo, Report No 620, 1978.
- [91] Kristensen, B. B., Madsen, O. L., Møller-Pedersen, B., and Nygaard, K.: *BETA Examples (Corresponding to the BETA Language Proposal as of August 1979)*, DAIMI IR-15, 1979.
- [92] Kristensen, B. B., Madsen, O. L., Møller-Pedersen, B., and Nygaard, K.: *BETA Language Proposal as of April 1979*, Norwegian Computing Center, Oslo, NCC Report No 635, 1979.
- [93] Kristensen, B. B., Madsen, O. L., Møller-Pedersen, B., and Nygaard, K.: *A Survey of the BETA Programming Language*, Norwegian Computing Center, Oslo, Report No 698, 1981.
- [94] Kristensen, B. B., Madsen, O. L., Møller-Pedersen, B., and Nygaard, K.: *Syntax Directed Program Modularization*, European Conference on Integrated Interactive Computing Systems, ECICS 82, Stresa, Italy, 1982, North-Holland.
- [95] Kristensen, B. B., Madsen, O. L., Møller-Pedersen, B., and Nygaard, K.: *Abstraction Mechanisms in the BETA Programming Language*, Tenth ACM Symposium on Principles of Programming Languages, Austin, Texas, 1983.
- [96] Kristensen, B. B., Madsen, O. L., Møller-Pedersen, B., and Nygaard, K.: *An Algebra for Program Fragments*, ACM SIGPLAN Symposium on Programming Languages and Programming Environments, Seattle, Washington, 1985.
- [97] Kristensen, B. B., Madsen, O. L., Møller-Pedersen, B., and Nygaard, K.: *Multisequential Execution in the BETA Programming Language*, Sigplan Notices, vol. 20, 1985.
- [98] Kristensen, B. B., Madsen, O. L., Møller-Pedersen, B., and Nygaard, K.: *Specification and Implementation of Specialized Languages*, unpublished manuscript, 1985.
- [99] Kristensen, B. B., Madsen, O. L., Møller-Pedersen, B., and Nygaard, K.: *Dynamic Exchange of BETA Systems*, unpublished manuscript, 1986.
- [100] Kristensen, B. B., Madsen, O. L., Møller-Pedersen, B., and Nygaard, K.: *The BETA Programming Language*, in *Research Directions in Object Oriented Programming*, Shriver, B. D. and Wegner, P., Eds.: MIT Press, 1987.
- [101] Kristensen, B. B., Madsen, O. L., Møller-Pedersen, B., and Nygaard, K.: *Classification of Actions or Inheritance Also for Methods*, ECOOP'87 – European Conference on Object-Oriented Programming, Paris, 1987, Lecture Notes in Computer Science vol. 276, Springer Verlag.
- [102] Kristensen, B. B., Madsen, O. L., Møller-Pedersen, B., and Nygaard, K.: *Object-Oriented Programming in the BETA Programming Language—Draft*, 1989.
- [103] Kristensen, B. B. and May, D. C. M.: *Activities: Abstractions for Collective Behavior*, ECOOP'96 – European Conference on Object-Oriented Programming, Linz, 1996, Lecture Notes in Computer Science vol. 1098, Springer Verlag.
- [104] Kristensen, B. B. and Østerbye, K.: *Conceptual Modeling and Programming Languages* SIGPLAN Notices, vol. 29, 1994.
- [105] Kristensen, B. B. and Østerbye, K.: *Roles: Conceptual Abstraction Theory and Practical Language Issues*, Theory and Practice of Object Systems (TAPOS), 1996, vol. 2(3).
- [106] Krogdahl, S.: *On the Implementation of BETA*, Norwegian Computing Center, Oslo, 1979.
- [107] Krogdahl, S.: *The Birth of Simula*, IFIP WG9.7 First Working Conference on the History of Nordic Computing (HiINC1), Trondheim, 2003, Springer.
- [108] Lakos, C.: *The Object Orientation of Object Petri Nets*, First international workshop on Object-Oriented Programming and Models of Concurrency—16th International Conference on Application and Theory of Petri Nets, 1995.
- [109] Liskov, B. H. and Zilles, S. N.: *Programming with Abstract Data Types*, ACM Sigplan Notices, vol. 9, 1974.
- [110] MacLennan, B. J.: *Values and Objects in Programming Languages*, SIGPLAN Notices, vol. 17, 1982.
- [111] Madsen, O. L.: *Block Structure and Object Oriented Languages*, in *Research Directions in Object Oriented Programming*, Shriver, B. D. and Wegner, P., Eds.: MIT Press, 1987.

- [112] Madsen, O. L.: *The implementation of BETA*, in *Object-Oriented Environments—The Mjølner Approach*: Prentice Hall, 1993.
- [113] Madsen, O. L.: *Open Issues in Object-Oriented Programming*, Software Practice & Experience, vol. 25, 1995.
- [114] Madsen, O. L.: *Semantic Analysis of Virtual Classes and Nested Classes*, OOPSLA'99 – Object-Oriented Programming, Systems Languages and Applications, Denver, Colorado, 1999, Sigplan Notices vol. 34, ACM Press.
- [115] Madsen, O. L., Magnusson, B., and Møller-Pedersen, B.: *Strong Typing of Object-Oriented Languages Revisited*, Joint OOPSLA/ECOOP'90 – Conference on Object-Oriented Programming, Systems, Languages, and Applications & European Conference on Object-Oriented Programming, Ottawa, Canada, 1990, Sigplan Notices vol. 25, ACM Press
- [116] Madsen, O. L. and Møller-Pedersen, B.: *What Object-Oriented Programming May Be—and What It Does Not Have to Be*, ECOOP'88 – European Conference on Object-Oriented Programming, Oslo, Norway, 1988, Lecture Notes in Computer Science vol. 322, Springer Verlag.
- [117] Madsen, O. L. and Møller-Pedersen, B.: *Virtual Classes—A Powerful Mechanism in Object-Oriented Programming*, OOPSLA'89 – Object-Oriented Programming, Systems Languages and Applications, New Orleans, Louisiana, 1989, Sigplan Notices vol. 24, ACM Press.
- [118] Madsen, O. L. and Møller-Pedersen, B.: *Part Objects and Their Location*, TOOLS'92: Technology of Object-Oriented Languages and Systems, Dortmund, 1992, Prentice Hall.
- [119] Madsen, O. L., Møller-Pedersen, B., and Nygaard, K.: *Object-Oriented Programming in the BETA Programming Language*: Addison Wesley, 1993.
- [120] Madsen, O. L. and Nørgaard, C.: *An Object-Oriented Metaprogramming System*, Hawaii International Conference on System Sciences, Hawaii, 1988, CRC Press.
- [121] Magnusson, B.: *Code Reuse Considered Harmful*, JOOP – Journal of Object-Oriented Programming, vol. 4, 1991.
- [122] Magnusson, B.: *Simula Runtime System Overview*, in *Object-Oriented Environments—The Mjølner Approach*: Prentice Hall, 1993.
- [123] Maier, C. and Moldt, D.: *Object Coloured Petri Nets—A Formal Technique for Object Oriented Modelling*, in *Concurrent Object-Oriented Programming and Petri Nets: Advances in Petri Nets*, vol. 2001/2001, *Lecture Notes in Computer Science*: Springer, 2001.
- [124] May, D. C.-M., Kristensen, B. B., and Nowack, P.: *Conceptual Abstraction in Modeling with Physical and Informational Material*, in *Applications of Virtual Inhabited 3D Worlds*, Qvortrup, L., Ed.: Springer Press, 2004.
- [125] Meyer, B.: *Eiffel: The Language*: Prentice Hall, 1992.
- [126] Mezini, M. and Ostermann, K.: *Conquering Aspects with Caesar*, AOSD'03, Boston, USA, 2003.
- [127] Møller-Pedersen, B., Belsnes, D., and Dahle, H. P.: *Rationale and Tutorial on OSDL: An Object-Oriented Extension of SDL*, SDL Forum'87 – State of the Art and future Trends, 1987, North-Holland.
- [128] Møller-Pedersen, B., Belsnes, D., and Dahle, H. P.: *Rationale and Tutorial on OSDL: An Object-Oriented Extension of SDL*, Computer Networks, vol. 13, 1987.
- [129] Naur, P.: *Revised Report on The Algorithmic Language ALGOL 60*, Communications of the ACM, vol. 6, 1963.
- [130] Nygaard, K.: *System Description by SIMULA—an Introduction*, Norwegian Computing Center, Oslo, S-35, 1970.
- [131] Nygaard, K.: *GOODS to Appear on the Stage*, ECOOP'97 – European Conference on Object-Oriented Programming, Jyväskylä, Finland, 1997, Lecture Notes in Computer Science vol. 1241, Springer Verlag.
- [132] Nygaard, K.: *Ole-Johan Dahl*, in *Journal of Object Technology*, vol. 1, 2002.
- [133] Odersky, M.: *The Scala Experiment—Can We Provide Better Language Support for Component Systems?*, Symposium on Principles of Programming Languages (POPL), Charleston, South Carolina, 2006.
- [134] Odersky, M., Altherr, P., Cremet, V., Emir, B., Maneth, S., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E., and Zenger, M.: *An Overview of the Scala Programming Language*, Ecole Polytechnique Fédérale de Lausanne, 1015 Lausanne, SwitzerlandIC/2004/64, 2004.
- [135] Odersky, M. and Zenger, M.: *Scalable Component Abstractions*, OOPSLA 2005 – Object-Oriented Programming, Systems Languages and Applications, San Diego, CA, USA, 2005, Sigplan Notices ACM Press.
- [136] Olsen, A., Færgemand, O., Møller-Pedersen, B., Reed, R., and Smith, J. R. W.: *Systems Engineering Using SDL-92*: North-Holland, 1994.
- [137] Palme, J.: *New Feature for Module Protection in SIMULA*, Sigplan Notices, vol. 11, 1976.
- [138] Palsberg, J. and Schwartzbach, M. I.: *Type Substitution for Object-Oriented Programming*, Joint OOPSLA/ECOOP'90 – Conference on Object-Oriented Programming: Systems, Languages, and Applications & European Conference on Object-Oriented Programming, Ottawa, Canada, 1990, ACM SIGPLAN Notices vol. 25, ACM Press New York.
- [139] Palsberg, J. and Schwartzbach, M. I.: *Object-Oriented Type Inference*, OOPSLA'91 – Conference on Object-Oriented Programming: Systems, Languages, and Applications, Phoenix, Arizona, 1991, SIGPLAN Notices vol. 26, ACM Press.
- [140] Palsberg, J. and Schwartzbach, M. I.: *Object-Oriented Type-Systems*: John Wiley & Sons, 1994.
- [141] Sandvad, E.: *Object-Oriented Development—Integrating Analysis, Design and Implementation*, Computer Science Department, University of Aarhus, Aarhus, DAIMI PB-302, 1990.
- [142] Sandvad, E.: *An Object-Oriented CASE Tool*, in *Object-Oriented Environments—The Mjølner Approach*, Knudsen, J. L., Löfgren, M., Madsen, O. L., and Magnusson, B., Eds.: Prentice Hall, 1994.

- [143] Sandvad, E., Grønbæk, K., Sloth, L., and Knudsen, J. L.: *A Metro Map Metaphor for Guided Tours on the Web: the Webvise Guided Tour System*, The Tenth International World Wide Web Conference, Hong Kong, 2001.
- [144] Seligmann, J. and Grarup, S.: *Incremental Mature Garbage Collection Using the Train Algorithm*, ECOOP'95 – European Conference on Object-Oriented Programming, Aarhus, Denmark, 1995, Lecture Notes in Computer Science vol. 952, Springer Verlag.
- [145] Shriver, B. D. and Wegner, P.: *Research Directions in Object Oriented Programming*: MIT Press, 1987.
- [146] Smith, B.: *Personal Communication*, Madsen, O. L., Ed. Stanford, 1984.
- [147] Smith, J. M. and Smith, D. C. P.: *Database Abstraction: Aggregation and Generalization*, ACM TODS, vol. 2(2), 1977.
- [148] Stroustrup, B.: *The C++ Programming Language*, 2000.
- [149] Sørgaard, P.: *Object-Oriented Programming and Computerised Shared Material*, ECOOP'88 – European Conference on Object-Oriented Programming, Oslo, Norway, 1988, Lecture Notes in Computer Science vol. 322, Springer Verlag.
- [150] Thomsen, K. S.: Multiple Inheritance, a Structuring Mechanism for Data, Processes and Procedures, Master thesis, Department of Computer Science, Aarhus University, Aarhus 1986.
- [151] Thomsen, K. S.: *Inheritance on Processes, Exemplified on Distributed Termination Detection*, International Journal of Parallel Programming, vol. 16(1), pp. 17–52, 1987.
- [152] Thorup, K. K.: *Genericity in Java with Virtual Types*, ECOOP'97 – European Conference on Object-Oriented Programming, Jyväskylä, Finland, 1997, Lecture Notes in Computer Science vol. 1241, Springer Verlag.
- [153] Thorup, K. K. and Torgersen, M.: *Unifying Genericity—Combining the Benefits of Virtual Types and Parameterized Classes*, ECOOP'99 – European Conference on Object-Oriented Programming, Lisbon, Portugal, 1999, Lecture Notes In Computer Science vol. 1628, Springer Verlag.
- [154] Torgersen, M.: *Virtual Types are Statically Safe*, 5th Workshop on Foundations of Object-Oriented Languages, San Diego, CA, USA, 1998.
- [155] Torgersen, M., Hansen, C. P., Ernst, E., von der Ahé, P., Bracha, G., and Gafter, N.: *Adding Wildcards to the Java Programming Language*, SAC, Nicosia, Cyprus, 2004, ACM Press.
- [156] Ungar, D. and Smith, R. B.: *Self: The Power of Simplicity*, OOPSLA'87 – Object-Oriented Programming Systems, Languages and Applications, Orlando, Florida, USA, 1987, Sigplan Notices vol. 22, ACM Press.
- [157] Vaucher, J.: *Prefixed Procedures: A Structuring Concept for Operations*, IN-FOR, vol. 13, 1975.
- [158] Vaucher, J. and Magnusson, B.: *SIMULA Frameworks: the Early Years*, in *Object-Oriented Application Frameworks*, Fayad, M. and Johnsson, R., Eds.: Wiley, 1999.
- [159] Wegner, P.: *On the Unification of Data and Program Abstraction in Ada*, Tenth ACM Symposium on Principles of Programming Languages, Austin, Texas, 1983, ACM Press.
- [160] Wegner, P. and Zdonick, S.: *Inheritance as an Incremental Modification Mechanism or What Like is and Isn't Like*, ECOOP'88 – European Conference on Object-Oriented Programming, Oslo, Norway, 1988, Lecture Notes in Computer Science vol. 322, Springer Verlag.
- [161] Wirth, N.: *The Programming Language Pascal*, Acta Informatica, vol. 1, 1971.
- [162] Yourdon, E. and Constantine, L. L.: *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*: Yourdon Press Computing Series, 1979.
- [163] Østerbye, K.: *Parts, Wholes, and Subclasses*, 1990 European Simulation Multi-conference, 1990.
- [164] Østerbye, K.: *Associations as a Language Construct*, TOOLS'99, Nancy, 1999.
- [165] Østerbye, K. and Kreutzer, W.: *Synchronization Abstraction in the BETA Programming Language*, Computer Languages, vol. 25 pp. 165–187, 1999.
- [166] Østerbye, K. and Olsson, J.: *Scattered Associations in Object-Oriented Modeling*, NWPER'98, Nordic Workshop on Programming Environment Research, Bergen, Norway, 1998, Informatics report 152, Department of Informatics, University of Bergen.
- [167] Østerbye, K. and Quistgaard, T.: *Framework Design Using Inner Classes—Can Languages Cope?*, Library Centric Software Design Workshop '05 in connection with OOPSLA 2005, San Diego, 2005.

Appendix: Time line

Below is a time line showing when events in the BETA project took place. After each event, the number of the section describing the event is shown. In some electronic versions of this paper there may be links to these sections as well as the part of the text describing the events.



The Development of the Emerald Programming Language

Andrew P. Black

Portland State University

black@cs.pdx.edu

Norman C. Hutchinson

University of British Columbia

norm@cs.ubc.ca

Eric Jul

University of Copenhagen

eric@diku.dk

Henry M. Levy

University of Washington

levy@cs.washington.edu

Abstract

Emerald is an object-based programming language and system designed and implemented in the Department of Computer Science at the University of Washington in the early and mid-1980s. The goal of Emerald was to simplify the construction of distributed applications. This goal was reflected at every level of the system: its object structure, the programming language design, the compiler implementation, and the run-time support.

This paper describes the origins of the Emerald group, the forces that formed the language, the influences that Emerald has had on subsequent distributed systems and programming languages, and some of Emerald's more interesting technical innovations.

Categories and Subject Descriptors D.3.0 [*Programming Languages*]: General; D.3.2 [*Language Classifications*]: Object-oriented languages; D.3.3 [*Language Constructs and Features*]: Abstract data types, Classes and objects, Inheritance, Polymorphism

General Terms abstract types, distributed programming, object mobility, object-oriented programming, polymorphism, remote object invocation, remote procedure call.

Keywords call-by-move, Eden, Emerald, mobility, type conformity, Washington

1. Introduction

Emerald was one of the first languages and systems to support distribution explicitly. More importantly, it was the first

Permission to make digital/hard copy of part of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date of appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Permission may be requested from the Publications Dept., ACM, Inc., 2 Penn Plaza, New York, NY 11201-0701, USA, fax:+1(212) 869-0481, permissions@acm.org

©2007 ACM 978-1-59593-766-7/2007/06-ART11 \$5.00

DOI 10.1145/1238844.1238855

<http://doi.acm.org/10.1145/1238844.1238855>

language to propose and implement the notion of *object mobility* in a networked environment: Emerald objects could move around the network from node to node as a result of programming language commands, while they continued to execute. Object mobility was supported by location-independent object addressing, which made the location of the target of an object invocation semantically irrelevant to other objects, although facilities were provided for placing objects on particular machines when required. At a high level, Emerald invocation could be thought of as an early implementation of remote procedure call (RPC) [14], but with a much more flexible and dynamic binding system that allowed an object to move from one node to another between (and during) invocations of methods. Furthermore, as seen from a programmer's point of view, Emerald removed the "remote" from "remote procedure call": the programmer did *not* have to write any additional code to invoke a remote object compared to a local object. Instead, all binding, marshaling of parameters, thread control, and other tedious work was the responsibility of the implementation, i.e., the compiler and the run-time system.

In addition, Emerald sought to solve a crucial problem with distributed object systems at the time: terrible performance. Smalltalk had pioneered an extremely flexible form of object-oriented programming, but at the same time had sacrificed performance. Our stated goal was local performance (within a node) competitive with standard programming languages (such as C), and distributed performance competitive with RPC systems. This goal was achieved by our implementation.

1.1 Ancient History

Emerald forms a branch in a distributed systems research tree that began with the Eden project [3] at the University of Washington in 1979. Setting the context for Emerald requires some understanding of Eden and also of technology at that time. In 1980, at the start of the Eden project, local area networks existed only in research labs. Although early Eth-

ernet [76] systems had been in use at Xerox PARC for some time, the industrial Ethernet standard had only recently been completed and the first products were still in development. To the extent that network applications existed at all, outside of PARC, they were fairly primitive point-to-point applications such as FTP and email. UNIX did not yet support a socket abstraction, and programming networked applications required explicit message passing, which was difficult and error-prone.

Eden was a research project proposed by a group of faculty at the University of Washington (Guy Almes, Mike Fischer, Helmut Golde, Ed Lazowska, and Jerre Noe). The project received one of the first grants from what seemed even at the time to be a very far-sighted program at the National Science Foundation called Coordinated Experimental Research (CER), a program that sought to build significant expertise and infrastructure at a number of computer science departments around the United States. The stated goal of Eden was to support “integrated distributed computing.” While nobody had distributed systems at the time, it was clear that someday in the future such systems would be commonplace, and that programming them using existing paradigms would be extremely difficult. The key insight of the Eden team was to use objects—a computational model and technology that was still controversial in both the operating system and the programming language worlds at that time.

Eden was itself a descendant of Hydra [108], the father of all object-based operating systems, developed by Bill Wulf and his team at Carnegie-Mellon in the early 1970s. Guy Almes, a junior faculty member at UW, had written his thesis on Hydra at CMU [4], and brought those ideas to Washington. The idea behind an object-based distributed system was that every resource in the network—a file, a mail message, a printer, a compiler, a disk—would be an object. At the time we looked on an object as nothing more than some data bound together with program code that operated on that data to achieve a specific goal, such as *printing* a file object, or *sending* a mail message object, or *running* a compiler object on a file object. The code was partitioned into what are now referred to as *methods* for printing, sending, running, etc., but at the time we just thought of the code as a bunch of procedures that implemented operations on the object’s data.

The great thing about objects was that client programs could use an object simply by invoking an operation in that object and supplying appropriate parameters; they didn’t have to concern themselves with how the object was implemented. Thus, objects were a physical realization of Parnas’ principle of information hiding [85]. The key insight behind distributed object-based computing was that the same principle applied to the *location* of the object: a client could use an object without worrying about where that object was actually located in the network, and two conceptually similar objects (for example, two files) that were located in disparate places

might have completely different implementations. The Eden system would take care of finding the object, managing the remote communication, and invoking the right code, all invisibly to the programmer.

Several other research projects also explored the distributed object notion, notably Argus at MIT [71] and Clouds at Georgia Tech [2]. Argus, as a language design, was essentially complete before the Emerald project started. Thus, Argus was contemporary with Eden rather than with Emerald, and indeed shared with Eden the idea that there were two kinds of objects—“large” objects that were remotely accessible (and, in the case of Argus, transactional), and small local objects (essentially, CLU objects). Some of the Clouds ideas moved to Apollo Computer with the principals, and appeared in the Apollo Domain system; there were several visits between Apollo and Eden personnel.

This distributed object model is now the dominant paradigm for Internet programming, whether it is Java/J2EE, Microsoft .NET, CORBA, SOAP, or whatever. We take it for granted. So it is hard to convey how controversial the distributed object idea was in the early 1980s. People thought that distributed objects would never work, would be way too slow, and were just dumb. Objects had not yet been accepted even in the non-distributed world: Simula was not mainstream, C++ would not be invented for some years [100], and Java wouldn’t appear for a decade and a half. Smalltalk had just been released by PARC and was gathering a following amongst computing “hippies,” but unless one had a Dorado, it was not fast enough for “real” applications. Alan Borning, our colleague at Washington, had distributed copies of the influential 1981 *Smalltalk* issue of *Byte* magazine, but in our opinion, the focus of Smalltalk was on flexibility to the detriment of performance. Smalltalk, although an impressive achievement, contributed to the view that poor performance was inherent in object-oriented languages.

Emerald was a response to what we had learned from our early experience with Eden and from other distributed object systems of that time. In fact, it was a follow-on to Eden before Eden itself was finished. Before describing that experience and its implications, we discuss the team and their backgrounds and the beginnings of the project.

1.2 The People and the Beginning of Emerald

The Emerald group included four people:¹

- Andrew Black joined UW and the Eden project as a Research Assistant Professor in November 1981. He had a background in language design from his D.Phil. at Oxford (under C.A.R. Hoare), including previous work on concurrency and exception handling. Andrew brought

¹ Another Ph.D. student, Carl Binding, participated in some of the initial discussions with a view to being responsible for a reasonable GUI for Emerald; he decided to do a different Ph.D., so the GUI of the Emerald system remained quite primitive.

the perspective of a language designer to Eden, which had been exclusively a “systems” project.

- Eric Jul came to UW as a Ph.D. student in September 1982 with a Master’s degree in Computer Science and Mathematics from the University of Copenhagen. He had previous experience with Simula 67 [15, 41] and Concurrent Pascal [29, 30] at the University of Copenhagen, where he had ported Concurrent Pascal to an Intel 8080 based microcomputer [96], and also written a master’s thesis [57] that described his implementation of a small OS whose device drivers were written entirely in Concurrent Pascal.
- Norm Hutchinson also came to UW in September 1982, having graduated from the University of Calgary. He had spent the previous summer on an NSERC-funded research appointment implementing a compiler for Simula supervised by Graham Birtwistle, an author of *SIMULA Begin* [15] and an object pioneer.
- Henry (Hank) Levy had spent a year at UW in 1980 on leave from Digital Equipment Corp., where he had been a member of the VAX design and implementation team. While at UW he was part of the early Eden group and also wrote an MS thesis on capability-based architectures, which eventually became a Digital Press book [70]. Hank rejoined UW as a Research Assistant Professor in September 1983, and brought with him a lot of systems-building and architecture experience from DEC.

Shortly after his return to UW in 1983, Hank attended a meeting of the Eden group in which Eric and Norm gave talks on their work on the project. Although Hank was a coauthor of the original Eden architecture paper [67], Hank wasn’t up to date on the state of the Eden design and implementation, and hearing about it after being away for two years gave him more perspective. Several things about the way the system was built—and the way that it performed—did not seem right to him. After the meeting, Hank invited Eric and Norm to a coffee shop on “the Ave”, the local name for University Way NE, close by the UW campus.

At the meeting, the three of them discussed some of the problems with Eden and the difficulties of programming distributed applications. Hank listened to Eric’s and Norm’s gripes about Eden, and then challenged them to describe what they would do differently. Apparently finding their proposals reasonable, Hank then asked, “Why don’t you do it?”: thus the Emerald effort was born.

1.3 The Eden System

The problems with Eden identified at the coffee shop on the Ave were common to several of the early distributed object systems. Eden applications (that is, distributed applications that spanned a local-area network) were written in the Eden Programming Language (EPL) [21]—a version of Concurrent Euclid [50] to which the Eden team had added support

for remote object invocation. However, that support was incomplete: while making a remote invocation in Eden was much easier than sending a message in UNIX, it was still a lot of work because the EPL programmer had to participate in the implementation of remote invocations by writing much of the necessary “scaffolding”. For example, at the invoking end the programmer had to manually check a status code after every remote invocation in case the remote operation had failed. At the receiving end the programmer had to set up a thread to wait on incoming messages, and then explicitly hand off the message to an (automatically generated) dispatcher routine that would unpack the arguments, execute the call, and return the results. The reason for the limited support for remote invocation was that, because none of the Eden team had real experience with writing distributed applications, we had not yet learned what support should be provided. For example, it was not clear to us whether or not each incoming call should be run in its own thread (possibly leading to excessive resource contention), whether or not all calls should run in the same thread (possibly leading to deadlock), whether or not there should be a thread pool of a bounded size (and if so, how to choose it), or whether or not there was some other, more elegant solution that we hadn’t yet thought of. So we left it to the application programmer to build whatever invocation thread management system seemed appropriate: EPL was partly a language, and partly a kit of components. The result of this approach was that there was no clear separation between the code of the application and the scaffolding necessary to implement remote calls.

There was another problem with Eden that led directly to the Emerald effort. While Eden provided the abstraction of location-independent invocation of mobile distributed objects, the implementation of both objects and invocation was heavy-weight and costly. Essentially, an Eden object was a full UNIX process that could send and receive messages. The minimum size of an Eden object thus was about 200-300 kBytes—a substantial amount of memory in 1984. This clearly precluded using Eden objects for implementing anything “small” such as a syntax tree node. Furthermore, if two Eden objects were co-located on the same machine, the invocation of one by the other would still require inter-process communication, which would take hundreds of milliseconds—slow even by the standards of the day. In fact, things were even worse than that, because in our prototype the Eden “kernel” itself was another UNIX process, so sending an invocation message would require *two* context switches even in the local case, and receiving the reply another two. This meant that the cost of a single invocation between two Eden objects located on the same machine was close to half the cost of a remote call (137 ms vs. 300 ms). The good news was that Eden objects enjoyed the benefits of location independence: an object did not have to know whether the target of an invocation was on the same com-

puter or across the network. The bad news was that if the invoker and the target were on the same computer, the cost was excessive.

Because of the high cost of Eden objects, both in terms of memory consumption and execution time, another kind of object was used in EPL programs—a light-weight language-level object, essentially a heap data structure as implemented in Concurrent Euclid. These objects were supported by EPL’s run-time system. EPL’s language-level objects could not be distributed, i.e., they existed within the address space of a single Eden object (UNIX process) and could not be referenced by, nor moved to, another Eden object. However, “sending messages” between these EPL objects was extremely fast, because they shared an address space and an invocation was in essence a local procedure call.

The disparity in performance between local invocations of EPL objects and Eden objects was huge—at least three orders of magnitude.² This difference caused programmers to limit their use of Eden objects to only those things that they felt absolutely needed to be distributed. Effectively, programmers were using two different object semantics—one for “local” objects and one for “remote” objects. Worse, they had to decide *a priori* which was which, and in many cases, needed to write two implementations for a single abstraction, for example, a local queue and a distributed queue. In part, these two different kinds of objects were a natural outgrowth of the two (unequal) thrusts of the Eden project: the primary thrust was implementing “the system”; providing a language on top was an afterthought that became a secondary goal only after Andrew joined the project. However, in part the two object models were a reflection of the underlying implementation: there were good engineering reasons for the two *implementations* of objects.

The presence of these two different object models was one of the things that had bothered Hank in the Eden meeting that he had attended. In their discussions, Eric, Norm and Hank agreed that while the two *implementations* made sense, there was no good reason for this implementation detail to be visible to the programmer. Essentially, they wanted the language to have a *single* object abstraction for the programmer; it would be left to the *compiler* to choose the most appropriate implementation based on the way that the object was used.

1.4 From Eden to Oz

The result of the meeting on the Ave was an agreement between Eric, Hank and Norm to meet on a regular basis to discuss alternatives to Eden—a place that Hank christened the land of Oz, after the locale of L. Frank Baum’s fantasy story [9]. A memo *Getting to Oz* dated 27 April 1984 (reference [69], included here as Appendix A) describes how the discussions about Oz first focused on low-level kernel issues:

² A local invocation in Eden took 137 ms in October 1983, while a local procedure call took less than 20 μ s.

processes and scheduling, the use of address spaces, and so on. This led to the realization that compiler technology was necessary to get adequate performance: rather than calling system service routines to perform dynamic type-checking and to pack up data for network interchange, a smart compiler could perform the checking statically and lay down the data in memory in exactly the right format.

The memo continues:

It is interesting that up to this point our approach had been mostly from the kernel level. We had discussed address spaces, sharing, local and remote invocation, and scheduling. However, we began to realize more and more that the kernel issues were not at the heart of the project. Eventually, we all agreed that language design was the fundamental issue. Our kernel is just a run-time system for the Oz language (called Toto) and the interesting questions were the semantics supported by Toto.

Eventually the name Toto was dropped; Hank thought that it was necessary to have a “more serious” name for our language if we wanted our work to be taken seriously. For a time we used the name Jewel, but eventually settled on Emerald, a name that had the right connotations of quality and solidity but preserved our connection to the Land of Oz. Moreover, the nickname for Seattle, where Emerald was developed, is *The Emerald City*.

The Emerald project was both short and long. The core of the language and the system, designed and implemented by Eric and Norm for their dissertations, was completed in a little over 3 years, starting with the initial coffee-shop meeting in the autumn of 1983, and ending in February 1987 when the last major piece of functionality, process mobility, was fully implemented. However, this view neglects the considerable influence of the Eden project on Emerald, and the long period of refinement, improvement and derived work after 1987. Indeed, Niels Larsen’s PhD thesis on transactions in Emerald was not completed until 2006 [66]. Figure 1 shows some of the significant events in the larger Emerald project; we will not discuss them now, but suggest that the reader refer back to the figure when the chronology becomes confusing.

1.5 Terminology

A note on terminology may help clarify the remainder of the paper. The terms message and message send, introduced by Alan Kay to be consistent with the metaphor of an object as a little computer, were generally accepted in the object-oriented-language community. However, we decided not to adopt these terms: in the distributed-systems community, messages were things that were sent over networks between real, not metaphorical computers. We preferred the term *operation* for what Smalltalk (following Logo) called a method, and we used the term *operation invocation* for

Emerald Timeline

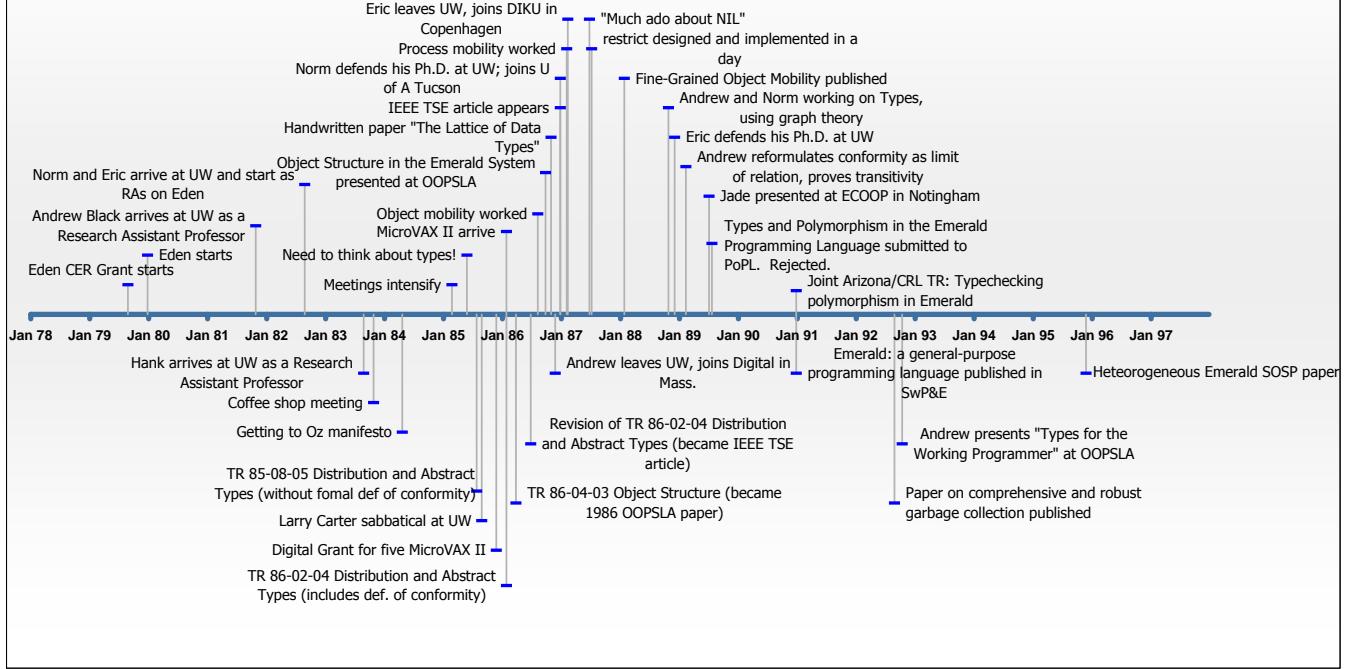


Figure 1: Some significant events in the Emerald Project

message send. In some ways this was unfortunate: language influences thought, and because we avoided the use of the terms message and message send, it was many years before some of us really understood the power of the messaging metaphor.

Emerald used the term *process* in the same sense as Concurrent Pascal and Concurrent Euclid. A process was light in weight and was protected by the language implementation rather than by an operating system address space; today the term *thread* is used to describe the same concept. Similarly, Emerald used the term *kernel* in the same sense as its predecessor languages: it meant the language-specific run-time support code that implemented object and process creation and process scheduling, and translated lower-level interrupts into operations on the language's synchronization primitives. Successor languages now often use the term *Virtual Machine* about such run-time support. Indeed, if we created Emerald today, we would have used the term Emerald Virtual Machine.

2. The Goals of Emerald

Beyond “Improving on Eden,” Emerald had a number of specific goals.

- *To implement a high-performance distributed object-oriented system.* While the Eden experience had convinced us that objects were indeed a good abstraction for writing distributed systems, it did nothing to dispel doubts about the performance of distributed objects. We believed that distributed objects could be made efficient and wanted to demonstrate this *efficiency goad* by construction. Norm saw that by exploiting compiler technology, we could not only make run-time operations more efficient, but could eliminate many of them altogether (by moving the work to compile time). Eric was already concerned with wide-area distribution: even during the planning phases of the project he was thinking of sending objects from Copenhagen to Seattle, although this did not become possible for several years citeFolmer93.

- *To demonstrate high-performance objects.* Stepping back from *distributed* objects, we were also concerned with validating the ideas of object-oriented programming *per se*. Smalltalk-80 had created a lot of excitement about the power and flexibility of objects, but there was also a lot of skepticism about whether or not objects could ever be made reasonably efficient. The failure of the Intel iAPX 432 architecture [79] effort had, in the minds of some people, reinforced the view that objects would

always impose excessive overhead. Dave Ungar’s dissertation research at Stanford was investigating how hardware could help overcome the inherent costs of supporting objects [102]. One of our goals was to show that operations similar to those found in a low-level language like C could be executed in Emerald with comparable efficiency. Thus, we wanted the cost of incrementing an integer variable to be no higher in Emerald than in C. Of course, the cost of invoking a *remote* object would be higher than the cost of a C procedure call. An important principle was that of *no-use, no-cost*: the additional cost of a given functionality, e.g., distribution, should be imposed only on those parts of the program that used the additional functionality.

- *To simplify distributed programming.* Although Eden was a big step forward with respect to building distributed applications, it still suffered from major usability problems. The Eden Programming Language’s dual object model was one, the need to explicitly receive and dispatch incoming invocation messages was another, and the need to explicitly check for failure and deal with checkpointing and recovery was yet another. One of the goals for Emerald was to make programming simpler by automating many of the chores that went along with distributed programming. If a task was difficult enough to make automation impossible, then the goal was to automate the bookkeeping details and thus free the programmer to deal with the more substantive parts of the task. For example, we had no illusion that we could automate the placement of objects in a distributed system, and left object location policy under the explicit control of the programmer. However, the mechanisms for creating the objects on, or moving them to, the designated network node, and of invoking them once there, could all be automated.
- *To exploit information hiding.* We believed in the principle of information hiding [85], and wanted our language to support it. The idea that a single concept in the language, with a single semantics—the object—might have multiple implementations follows directly from the principle of information hiding: the details of the implementation are being hidden, and the programmer ought to be able to rely on the interface to the object without caring about those details. In the *Getting to Oz* memo [69] we wrote: “Another goal, then, and a difficult one, is to design a language that supports both large objects (typically, operating system resources such as files, mailboxes, etc.) and small objects (typically, data abstractions such as queues, etc.) using a single semantics.”
- *To accommodate failure.* Emerald was intended for building distributed applications. What distinguishes distributed applications from centralized ones is that both the individual computers on which the application runs and the network links that connect them can fail; nev-

ertheless, the application as a whole should continue to run. We realized that handling failures was a natural part of programming a distributed application, in fact, failures were *anticipated* conditions that arose from distribution: computers and programs would crash and restart, network links would break, and consequently objects would become temporarily or permanently unavailable. We saw it as the programmer’s job to deal with these failures, and therefore as the language designer’s job to provide the programmer with appropriate tools. It was explicitly not a goal of Emerald to provide a full-blown exception-handling facility [69], or to impose on the programmer a particular way of handling failure, as Argus was doing with its pioneering support for transactions in distributed systems [71].

- *To minimize the size of the language.* We wanted Emerald to be as small and simple as possible while meeting our other goals. We planned to achieve this by “pulling out” many of the features that in other languages were built-in, and instead relying on the abstraction provided by objects to extend the language as necessary. Our goal was a simple yet powerful object-based language that could be described by a language report not substantially larger than that defining Algol 60 [8]. We did not want to go as far as Smalltalk and rely on library code for the basic control structures: this would have conflicted with our efficiency goal. We also felt that not allowing the programmer to define new control structures would not adversely affect the writing of distributed applications.

We did decide to build in only the simplest kind of fixed-length Vector, and to rely on library code to provide all the common data structures, such as arrays of variable size, and lists. This meant that the type system had to be adequate to support user-defined container objects; striving to do this led us to make several innovations in object-oriented type systems, even though inventing a type system was not one of our primary goals. The same idea of pulling out functionality into a library was applied to integer arithmetic. We had to decide whether to include infinite-precision or fixed-length integers. Infinite-precision arithmetic was clearly more elegant and would lead to simpler programs and a more concise language definition, but it would not help us in our quest to make $x \leftarrow x + 1$ as fast as in the C language. So we included fixed-precision integers in the base language, and allowed the programmer to define other kinds of integers in library code.

- *To support object location explicitly.* We knew that the placement of objects would be critical to system performance, and so our information-hiding goal could not be allowed to hide this performance-critical information from the programmer. So “we decided that the concept of object location should be supported by the language”

and that “the movement of objects should be easily expressible in the language” [69]. Early on we envisaged language support for determining the location of an object, moving the object to a particular host, and fixing an object at a particular host. The idea of *call-by-move* was floated in the 1984 *Getting to Oz* memo; call-by-move was a new parameter passing mechanism in which the invoker of an operation indicated explicitly that a parameter should be moved to the target object’s location. By March 1985 call-by-move was established, and Andrew was arguing for a separation of concerns between the language’s *semantics*, which he thought should be similar to those of any other object-based language (for example, all parameters would be passed by object reference), and its *locatics* — a term that we invented to mean the part of the language concerned with the placement of objects. We adopted this separation: the primitives that controlled the placement of objects were designed to be orthogonal to the ordinary operational semantics of the language. For the most part we were successful in maintaining this separation, but locatics does inevitably influence failure semantics. For example, if two objects are co-located, then an invocation between them can never give rise to a *object unavailable* event.

3. How We Worked

The Emerald group was informal and largely self-organized. Hank kept Eric and Norm in line by requiring regular meetings, minutes, and written progress reports. He was also the prime mover in getting the first external funding that the project received: a grant of five MicroVAX computers from Digital Equipment Corp. Andrew attended some of the meetings, and had many impromptu and as well as scheduled discussions with Norm about the language design, and in particular the type system.

Minutes from March 1985 reveal an intent to meet three times per week. They state: “It will be Norm’s job to see that local invocations execute as fast as local procedure calls, and Eric’s to make sure that remote invocations go faster than Eden.” This captured the major distribution of work: Norm implemented the compiler, while Eric worked on the run-time system, which we called the Emerald kernel.

They pursued an incremental implementation strategy. The first compiler was for a simplified version of the language and produced byte-codes that were then interpreted. The interpreter ran on top of a kernel that provided threads. Initially there was no I/O other than a `println` instruction provided by the interpreter. In this way, they very quickly had a working prototype, even though it could execute only the simplest of programs. Over time, the compiler was modified to generate VAX assembly language, which ran without needing interpretation, but which still called kernel procedures for run-time support. This incremental strategy brought new

functionality and better performance every day, and was a real catalyst for the development of the prototype. Looking back, we realize that we pioneered many of the techniques that have now become popular as part of agile development practices such as XP [10], although there is no evidence to suggest that those who invented XP were aware of what we had done.

A hallmark of Emerald was the close integration between the compiler and the run-time kernel. In part, this was achieved by putting the compiler writer (Norm) and the kernel implementor (Eric) in the same office, together with two workstations. (It was unusual at the time to allocate expensive workstations to graduate students.) Norm and Eric could clarify even minor details immediately. But there was also a technical side to this integration. The compiler and the kernel had to agree on several basic data structures. For example, the structure of an object descriptor was laid down in memory by the compiler, and manipulated by the kernel. The alternative approach in which this structure is encapsulated in a single kernel module would have simplified the software engineering, but it would also have meant that whenever the compiler wanted the code that it was generating to create an object, it would be forced to lay down code that made a call to the run-time kernel. This would have conflicted with our efficiency goal.

To ensure compatibility between the compiler and the kernel, all of these basic data structures were defined in a single file, both in C (for the compiler) and in assembly language (for the kernel). Careful use of the C preprocessor meant that this file could be treated either as a C program with assembly language comments, or as assembly language with C comments. The development of the shared description of the data structures was aided by the presence of a whiteboard where the important data structures were described; by convention, the truth was on the whiteboard, and the rest of the system was made to conform to it (see Figure 2). We avoided divergence by rebuilding the compiler and run-time system completely every night and during lunch breaks. Even language changes were handled efficiently. Because all existing Emerald programs were stored on Norm’s and Eric’s MicroVAX workstations, we could decide to change the syntax or semantics of the language and within hours change the compiler, the run-time system and *all existing Emerald programs* to reflect the modification. Typically, such changes were made before lunch so that a rebuild could happen over lunch with a full afternoon to verify that things still worked.

Initial development took place on the VAX 11/780 and two VAX 11/750s of the Eden project, which were called June, Beaver, and Wally. In December 1985 we obtained our grant of MicroVAX workstations from DEC; the workstations themselves arrived a few months later. Three of these (Roskilde, Taber, and Whistler) were used as personal development machines by Eric, Norm and Hank. The others were



Figure 2: The whiteboard that defined the basic data structures in the Emerald system, February 1986.

used remotely in distribution experiments. We used micro-benchmarks to find the fastest instruction sequences for common operations, and found that sometimes sequences of RISC operations were faster than the equivalent single CISC instruction. One example of this was copying a small number of bytes, where repeated reads and writes were faster than using the VAX block move (MOVC3) instruction.

The interpreter itself was written in C, but in a way that treated C as a macro-assembler. We would first identify the most efficient assembly-language sequence for the task at hand; for example, we found that non-null loops were more efficient if the termination test came at the end rather than at the beginning, because this avoided an unconditional branch. We then wrote the C code that we knew would generate the appropriate assembly language sequence, for example, using **if (B) do ... while (B)** rather than **while (B) ...**

Over time, more and more data structures were defined to enable the compiler to transmit information to the interpreter. These included a source-code line number map, which enabled us to map back from the program counter to the source code line for debugging, and *templates* for the stack and for objects, which enabled us to distinguish between object-references and non-references. The latter information was essential for implementing object mobility, because object references were implemented as local machine addresses, and had to be translated to addresses on the new host when

an object was moved. (The same information is of course useful for garbage collection, but the garbage collector was not implemented until later.)

By June 1985 we had realized that we needed to think seriously about types. We had decided that utility objects like variable-sized arrays and sets should not be built into the language, but would be provided in a library, as they were in Smalltalk. This would keep the language simple. However, we wanted these objects to have the same power, and offer the same convenience to programmers, as the built-in arrays and sets of Pascal. We had also decided early on that Emerald was to be statically typed, both for efficiency and to make possible the early detection and reporting of programming errors. We therefore needed a way of declaring that an array or a set would contain objects of a certain type—in other words, we needed parameterized types. We were influenced by the Russell type system [43], one of the few contemporary systems that supported parameterized types. We were also aware of CLU's decision to provide a parameterization mechanism that allowed procedures and clusters to be dependent on manifest values, including types; this mechanism (which used square brackets and operated at compile time) was completely separate from the mechanism used to pass arguments to procedures (which used parentheses and operated at run time). However, we realized that an open distributed system could not make a clear distinction between

compile time and run time: it is possible to compile new pieces of program and add them to the running system at any time. Thus, while we planned to check types statically whenever this was possible, we knew from the outset that sometimes we would have to defer type checking until run time, for the very good reason that the object that we needed to check arrived over the network at run time, having been compiled on a different computer. So we followed Russell, and made types first-class values that existed at run time and could be manipulated like any other values. In an object-oriented language, the obvious way to do that was to make types objects, so that is what we did in Emerald.

4. Technical Innovations

As we have discussed, the goals of Emerald centered on simplifying the programming of distributed applications. The way in which Emerald supports distribution has been described in a journal article [59] that was widely cited in the 1980s and 1990s but may not be familiar to this audience. In addition, Emerald made a number of other, lesser-known technical contributions that are nevertheless historically interesting. In this section we first summarize Emerald’s object model and then discuss a few of these other contributions.

4.1 Emerald’s Object Model

Key to any object-based language or system is its object model: the story of how its objects can be created and used, and what properties they enjoy. We have previously described the motivation for Emerald’s single object model: the problem with having both heavyweight and lightweight objects visible to the programmer, and the need for the programmer to choose between them. This was a reflection of the state of the art in the early 1980s, which recognized two very distinct object worlds: object-oriented programming languages, such as Smalltalk, and object-based operating systems, such as Eden and Hydra. In object-oriented languages, the conceptual model of an object was a small, lightweight data structure whose implementation was hidden from its clients. In object-oriented operating systems, the conceptual model of an object was a heavyweight, possibly remote operating system resource, such as a remote disk, printer, or file. Consequently, these systems frequently used an OS process to implement an object.

4.1.1 A Single Object Model

An important goal of Emerald was to bridge the gulf between these two worlds and to provide the best of each in a unified object model. On the one hand, we wanted an Emerald object to be as light in weight as possible — which meant as light as an object provided by a procedural programming language of the time. On the other hand, we wanted an Emerald object to be *capable* of being used in the OS framework, for example as a distributed resource that was transparently accessible across the local-area network.

Emerald achieved this goal. It provided a single, simple object model. An object, once created, lived forever and was unique, that is, there was only one instance of it at any time. Each object had (1) a globally unique *name*, (2) a *representation*, i.e., some internal state variables that contained primitive values and references to other objects, and (3) a set of *operations* (methods) that might be invoked on the object. Some of these operations could be declared to be *functional*, which meant that they computed a single value but had no effect and did not access any mutable state outside of the object itself. Any Emerald object could also have an associated *process*, which meant that an object could be either active (having its own process) or passive (executing only as a result of invocation by another process); the process model is described fully in Section 4.3.1.

4.1.2 Immutability

One of our more powerful insights was to recognize the importance of immutability in a distributed system. With ordinary, mutable objects, the distinction between the original of an object and a copy is very important. You can’t just go off and make a copy of an object and pass it off as the original, because changes to the original won’t be reflected in the copy. So programmers must be kept aware of the difference, and the object and the copy must have separate identities. However, with immutable objects, the copy and the original will always have the same value. If the language implementation lies and tells the programmer that the two objects have the same identity — that they are in fact one and the same — the programmer will never find out.

The idea of an immutable object was familiar to us from CLU. While this idea did at first seem strange — after all, one of the main distinctions between objects and values was that objects can change state — a little thought showed that it was inevitable. Even the most ardent supporter of mutable objects would find it difficult to argue that 3, π and ‘h’ should be mutable. So, the language designer is forced either to admit non-objects into the computational universe or to allow immutable objects. Once immutable primitive objects have been admitted, there are strong arguments in favor of allowing user-defined immutable objects, and little reason to protest. For example, Andrew knew that CLU’s designers had eventually found it expedient to provide two versions of all the built-in container objects, one mutable and the other immutable, even though they had initially decided to provide only mutable containers [92, 28].

Once we accepted that some objects were immutable, we found that there were many benefits to formally distinguishing immutable from mutable objects by introducing the **immutable** keyword. Emerald made wide use of the fact that some objects were known to be immutable. Although the language semantics said that there was a unique integer object 3, it would have been ridiculous to require that there really be only a single 3 in a distributed system. Because inte-

gers were immutable, the implementation was free to create as many copies of 3 as was expedient, and to insist (if anyone asked, by using the `==` operation to test object identity) that they were all the very same object. The implementation was lying to the user, of course, but it was lying so well and so consistently that it would never betray itself.

Another place where Emerald used immutability was for *code objects*. Logically, each Emerald object owned its own code, but if a thousand similar objects were created containing the same code, there was clearly no need to create a thousand copies of the code. But neither was it reasonable to insist that there could be only a single copy: in practice, the code needed to be on the same computer as the data that it manipulated. The obvious implementation was to put one copy of the code on each computer that hosted one of the objects; if an object were moved to a new computer, a copy of the code would be placed there too, unless a copy were there already. Because the code was immutable, we could pretend that all of these copies were logically the same, and this pragmatic use of copying by the implementation did not “show through” in the language semantics. Indeed, from the programmer’s point of view, code objects did not exist at all.

Immutability applied to programmer-defined objects as well as built-in objects. Indeed, any object could be declared as immutable by the programmer: immutability was an assertion that the *abstract* state of the object did not change. It was quite legal, for example, to maintain a cache in an immutable object for efficiency purposes, provided that the cache did not change the object’s abstract state.

Functional operations on immutable objects had the property that they always returned the same result. This meant that they could be evaluated early without changing the program’s semantics. In particular, functional operations on immutable objects could be evaluated at compile time: they were manifest. This turned out to be vitally important to Emerald’s type system, because we required types of variables and operations to be manifest expressions: the declarator `Array.of[Integer]` is an invocation of the function `of` on the immutable object `Array`, with the immutable object `Integer` as argument.

Our decision to trust the programmer to say when an object was immutable or an operation was functional, rather than attempting to enforce these properties in the compiler, arose from our experience with Concurrent Euclid (CE). Concurrent Euclid distinguished between functions and procedures, and in CE the compiler would enforce the distinction by emitting an error message if a function attempted to do something that had an effect. We had found it to be enormously frustrating to get an error message when we were trying to debug a program by putting a *print* statement in a function. (The compiler was smarter than we thought. We eventually found out that the error messages were actually warnings: the compiler generated perfectly good code de-

spite the existence of the *print* statement!) What we learned from this was that programmers resent tools that get in their way: a language should help programmers to express what they need to express, rather than always trying to second-guess them. Interestingly, in “A History of CLU” Liskov explains that she rejected the idea of immutability declarations exactly because the compiler would not have been able to check them [72, p. 484–5].

4.1.3 Objects Were Encapsulated

Because of our background in distributed systems, we took encapsulation much more seriously than did those who thought that encapsulation was just something that one did to get software engineering brownie points. The only way to access an object in Eden was to send a message to the machine that hosted it: there were no “back doors” that could be left ajar. In Emerald, because all of the objects that were co-located shared a single address space, it would have theoretically been possible to provide a back door through which a co-located object could sneak a look at some private data. But this would clearly be a really bad idea: such a back door would need to be slammed shut if the target object moved to another machines, and objects could move at any time. Moreover, as system implementors, we had to know *all* of the references leading into and out of each object, and we had to know that those references were used only through legitimate invocations of the object. This knowledge is what made mobility possible; if an object could somehow “cons up” a direct reference to the internal state of another object, then mobility would be next to impossible. Moreover, Emerald objects were concurrent (see Section 4.3), so it was necessary to ensure that any access to the internal state of an object was subject to some kind of synchronization constraint.

In the Emerald language, we indicated encapsulation by distinguishing between operations that could be invoked on an object from the outside (which were flagged with the **export** keyword), and operations that could be invoked only on the object itself. The compiler used the list of exported operations to determine the signature of the object: attempts to invoke other operations were not only forbidden by the type system, but would in any case not be supported at run time.

It is worthwhile comparing this approach to encapsulation to that taken by Java. In Java (and in C++), encapsulation is class-based rather than object based: a field or a method that is designated as private can in fact be accessed by any other object that happens to be implemented by the same class. However, access to fields of remote objects is not supported by Java RMI, and consequently Java can’t make local and remote objects look the same.

4.1.4 One Object Model, Three Implementations

Although many of the newer languages with which we were familiar—Alphard, CLU, Euclid, and so on—claimed to support “abstract data types”, we realized that most of them didn’t. To us, *abstract* meant that two different implementations of the *same* type could coexist, and that client code would be oblivious to the fact that they were different. Other languages actually supported *concrete* data types: only a single implementation of each data type was permitted.

If we took encapsulation seriously—and, as we have discussed in Section 4.1.3, distribution meant that we had to—then Emerald objects would be characterized by truly abstract types. This meant that so long as two objects had the same set of operations, client code would treat them identically, and the fact that they actually had different implementations would be completely hidden from the programmer. When the object implementation and the client code were compiled separately—this might mean separate in space as well as separate in time—the difference would also be hidden from the compiler.

However, when the implementation and the client code were *not* compiled separately, then the compiler could take advantage of the ability for multiple implementations of the same type to coexist. The same source code could be compiled into different representations, if the compiler decided that there was a reason to do so. Thus, the compiler could choose a customized representation that took advantage of the fact than an object was used in a restricted way. Norm designed the compiler to examine the object constructors in the code that it was compiling and to choose between two alternative implementations for the object under construction.

Global was the most general implementation. A global object could move to other machines and could be invoked by any other object regardless of location. References to a global object were implemented as pointers to an *object descriptor*.

Local was an implementation optimized for objects that could never be referenced from a remote machine. The compiler chose this implementation when it could ascertain that a reference to the object could never cross a machine boundary. For example, if object *A* defined an internal object *B* but never passed a reference to *B* outside of its own boundary, then the compiler knew³ that *B* could only be invoked by code in *A*. This allowed the compiler to strip away all code related to remote referencing, including the object descriptor.

There was actually a third implementation, *direct*, which we used to implement objects of “primitive” types (*Boolean*, *Character*, *Integer*, etc.). However, direct objects were a

bit of a cheat, because for these types we gave up on type abstraction. Primitive types are discussed in Section 4.2.5.

Letting the compiler choose the implementation meant that programs that did not use distribution could be compiled into code similar to that produced for a comparable C program. In particular, integer arithmetic and vector indexing were implemented by single machine instructions on the VAX. Moreover, we were able to provide multiple representations without boxing, and could thus represent integers by a direct bit pattern, just as in C. The result was that Emerald achieved performance very close to that of C—for the kind of simple non-distributed programs that could have been written in C. Nevertheless, the very same source code, used in a program that exploited distribution, could create remotely accessible distributed objects.

4.1.5 Object Constructors Replace Classes

During the initial development of Emerald we were unhappy about adopting Smalltalk’s idea of a class that could change dynamically. Smalltalk allowed a class to be modified, for example by adding an instance variable or a method, while instances of that class were alive; all of the instances immediately reflected the changes to their class. This worked well in a single-user centralized system, but we did not see how to adapt it to a distributed system. For performance reasons, the class would have to be replicated on every node that hosted an instance of the class; if the class were subsequently modified, we would be faced with the classic distributed update problem. The problem was compounded in Emerald by the fact that some machines might be inaccessible when the update occurred. For us, the semantics of class update seemed impossible to define in a satisfying manner: the only implementable semantics that we could think of was that the update would take effect *eventually*, but perhaps at widely differing times on different machines, and possibly far in the future.

The difficulty of defining update for classes was one of the reasons that we considered classes harmful.⁴ Another reason was that we wanted Emerald to be a simple language in which everything was an object. This implied that if we had classes they should also be objects, which would in turn need their own classes. Smalltalk’s metaclass hierarchy resolved this apparent infinite regression, but at the cost of significant complexity, which we also wanted to avoid.

We noted that in classic OO languages, such as Simula 67 and Smalltalk, the concept of class was used for several different purposes: as a classification scheme for object instances, as a template describing the internal structure of those instances, as a repository for their code, and as a factory for generating new instances [17, p. 85].

³ More precisely: the compiler could figure this out, using techniques from what is now called escape analysis.

⁴ During our car trip from Seattle to Portland for the first OOPSLA in 1986, we bounced around the idea of writing a paper about *Classes Considered Harmful*.

Because we had a type system, we did not need to use classes for classification. Code storage was managed by the kernel, because it was heavily influenced by distribution. To describe the internal structure of an instance, and to generate new instances, we invented the idea of an object constructor, based on the record constructor common in contemporary languages. An object constructor was an expression that, each time it was executed, generated a new object; the details of the internals of the object were specified by the constructor expression. Object constructors could create fully-initialized instances; in combination with block structure, this meant that the initial state of an object could depend on the parameters of the surrounding context. Moreover, because object constructors could be nested, it was easy to write an object that acted like a factory, that is, an object that would make other objects of a certain kind when requested to do so. It was also easy to write an object that acted like a prototype, in that it created a clone of itself when requested to do so.

4.1.6 Objects Are Not Fragmented

An important early decision was that the whole of an object would be located on a single machine. The alternative would have been to allow an object to be fragmented across multiple machines, but this would then require some other inter-machine communication mechanism (apart from operation invocation), thus making the language much larger. It would also seriously complicate reasoning about failure. We were also not convinced that fragmented objects were necessary: a distributed resource could after all be represented by a distributed collection of communicating (non-distributed) objects that held references to each other. One of the consequences of this design decision was that if any part of an object were locally accessible, then all other parts were also locally accessible. This allowed the compiler to strip away all distributed code for intra-object invocations and references.

4.1.7 The Unit of Mobility and the Concept of Attachment

Because we had decided that Emerald objects would be mobile, it might seem that we had also determined the unit of mobility: the object. Unfortunately, things were not quite that simple. An object contained nothing more than variables that were references to other objects, so moving an object to a new machine would achieve little: every time the code in the object invoked an operation on one of its variables, that operation would be remote.

If the variable referred to a small immutable object, it would clearly make sense to copy that object when the enclosing object was moved. Mutable objects could not be copied, but they could be *moved* along with the object that referenced them. A working memo written by Eric in August 1986 discusses various ways of defining groups of objects that would move together; the memo describes both imper-

ative and declarative language features. We eventually decided to introduce the concept of *attachment* by allowing the **attached** keyword to be applied to a variable declaration, with the meaning that the object referenced by the attached variable should be moved along with object containing the variable itself. Although this appeared to make attachment a static property, we realized that this was not in fact the case. An object could maintain two variables *a* and *u* that referenced the same object, *a* being attached and *u* not attached; by assigning *u* to *a*, or assigning **nil** to *a*, attachment could be controlled dynamically. Thus, the concept of attachment was powerful enough to implement dynamically composable groups efficiently. Joining or leaving a group could be as simple as a single assignment. Furthermore, in many cases when a new object joined a group, it would be assigned to a variable anyway, and so the additional cost would be zero. Attachment was also very simple to implement, because it required nothing more than a bit in the template that described the object's layout in memory (described in Section 4.6) and thus there was no per-object overhead, nor was there any cost for an object not using the concept.

4.2 The Emerald Type System

At the time we started the Emerald project, none of us knew very much about type theory, and innovation in type theory was not one of Emerald's explicit goals. However, we found that the goals that we had set for Emerald seemed to require features that were not mainstream; indeed, in 2007 some of them are still not mainstream. So we set about figuring out how to support the features we needed. The aim of this section is to describe that process and its end point; it draws from material originally written by Andrew and Norm in an unpublished paper dated 1989.

4.2.1 Emerald's Goals as They Relate to Types

We had decided from the first that Emerald needed the following features.

- Type declarations and compile-time type checking for improved performance. We had all done most of our programming in statically typed languages, and felt that Smalltalk's dynamic type checking was one of the reasons for its poor performance.
- A type system that was used for classification by behavior rather than implementation, or naming. This was demanded by our view that programming in Emerald consisted of adding new objects to an existing system of objects (as in Smalltalk) rather than writing standalone programs, (as in Euclid or Object Pascal). We might know nothing at all about the implementation of an object compiled elsewhere; we could demand that the object supported certain operations, but not that it had a particular implementation, or inherited from some other implementation.

- A small language in which utilities such as collection objects were not built-in, but could instead be implemented (and typed) in the language itself.

The second goal was most influential. The Smalltalk idea of operation-centric *protocols* formed the starting point for our work on what came to be known as abstract types and conformity-based typing.

4.2.2 The Purpose of Types

As we implied in Section 4.2.1, Emerald’s compile-time typing was more an article of faith than a carefully reasoned decision: we believed that compile-time typing would help the compiler to generate more efficient code. However, the decision to declare types in Emerald did not tell us what those declarations should mean. In a language like Pascal or Concurrent Euclid, a type declaration determined the data layout of the declared identifier. In Simula, a declaration determined the class of the object referenced by the identifier. But, as discussed in Sections 4.1.4 and 4.1.5, neither of these functions of types would be applicable in Emerald. So, what purpose should type declarations serve?

We turned for inspiration to two recently published papers on types. According to Donahue and Demers [44], the purpose of a type system was to prevent the misinterpretation of values—to ensure that the meaning of a program was independent of the particular representation of its data types. This independence would be necessary if we wished to abstract away from representation details—for example, so that some of them could be left to the compiler. Cardelli and Wegner [40] made the same point more colorfully:

A major purpose of type systems is to avoid embarrassing questions about representations, and to forbid situations in which these questions might come up. A type may be viewed as a set of clothes (or a suit of armor) that protects an underlying untyped representation from arbitrary or unintended use.

In fact, Donahue and Demers stated that a programming language was strongly typed exactly when it prevented this misinterpretation of values [44].

However, for the most part these papers discussed typing for values rather than objects. One of the properties of an object was encapsulation: the object’s own operations were the only ones that could be applied to its data. The only mechanism that was needed to enforce this encapsulation was static scoping as found in Simula 67 and Smalltalk. There was thus no need for Emerald’s type system to forbid “embarrassing questions about representations”: object encapsulation already did exactly this. So, should we abandon the idea of making Emerald statically typed? We thought not: we felt that a type system could usefully serve other goals. Amongst these goals were the classification of objects, earlier and more meaningful error detection and reporting, and improved performance.

In Smalltalk, objects were classified by class, and the inheritance relation between classes was important to understanding a Smalltalk program. However, classes conflated two issues: how the object behaved, and how it was implemented. Because Emerald took encapsulation seriously (see Section 4.1.3), we had to separate these issues. We decided that the programmer should use types to classify an object’s behavior, while implementation details should be left to the compiler. This led to the view of a type being the set of operations understood by the object, as discussed in Section 4.2.3.

A second problem we perceived with Smalltalk was that (almost) the only error that we saw was “message not understood”, and we didn’t see this error until run time. As programmers who had grown up with Pascal and Concurrent Euclid, when we accidentally added a boolean and an integer we expected an explicit *compile time* error message. We wanted to see similar error messages from the Emerald compiler whenever possible—which was quite often (see Section 4.2.4).

We also believed that at least some of Smalltalk’s performance problems were caused by the absence of static typing. We felt that if only the compiler had more information available to it about the set of operations that could be invoked on an object, it could surely optimize the process of finding the right code, called method lookup. We may have been right, although subsequent advances such as inline caches have largely eliminated the “lookup penalty”. The way that we used this extra information to eliminate method lookup is described in Section 4.2.5.

In summary, we came to the conclusion that there were three motivations for types in Emerald: to classify objects according to the operations that they could understand, to provide earlier and more precise error messages, and to improve the performance of method lookup. We now discuss in more detail the consequences of each of these motivations for types on the development of Emerald’s type system.

4.2.3 Types Were Sets of Operations

We were aware from our experience with Eden that a distributed system was never complete: it was always open to extension by new applications and new objects. Today, in the era of the Internet, the fact that the world is “under construction” has become a cliché, but in the early 1980s the idea that all systems should be extensible—we called it the “open world assumption”—was new.

A consequence of this assumption was that an Emerald program needed to be able to operate on objects that did not exist at the time that the program was written, and, more significantly, on objects whose *type* was not known when the application was written. How could this be? Clearly, an application must have *some* expectations about the operations that could be invoked on a new object, otherwise the appli-

cation could not hope to use the object at all. If an existing program P had minimal expectations of a newly injected object, such as requiring only that the new object accept the *run* invocation, many objects would satisfy those expectations. In contrast, if another program Q required that the new object understand a larger set of operations, such as *redisplay*, *resize*, *move* and *iconify*, fewer objects would be suitable.

We derived most of Emerald’s type system from the open world assumption. We coined the term *concrete type* to describe the set of operations understood by an actual, concrete object, and the term *abstract type* to describe the declared type of a piece of programming language syntax, such as an expression or an identifier. The basic question that the type system attempted to answer was whether or not a given object (characterized by a concrete type) supported enough operations to be used in a particular context (characterized by an abstract type). Whenever an object was bound to an identifier, which could happen when any of the various forms of assignment or parameter binding were used, we required that the concrete type of the object *conform* to the abstract type declared for the identifier. In essence, conformity ensured that the concrete type was “bigger” than the abstract type, that is, the object understood a superset of the required operations, and that the types of the parameters and results of its operations also conformed appropriately.

Basing Emerald’s type system on conformity distinguished it from contemporary systems such as CLU, Russell, Modula-2, and Euclid, all of which required equality of types. It also distinguished Emerald’s type system from systems in languages like Simula that were based on subclassing, that is, on the ancestry of the object’s implementation. In a distributed system, the important questions are not about the implementation of an object (which is what the subclassing relation captures) but about the operations that it implements.

For our purposes, type equality was not only unnecessary; it was counterproductive. Returning to the example above, there was no need to require that new objects presented to P supported *only* the operation *run*; no harm could come from the presence of additional operations, because P would never invoke them. The idea of conformity was that a type T conformed to a type U , written $T \diamond U$, exactly when T had all of U ’s operations, when the result types of those operations conformed, and when the argument types conformed *inversely*. We chose the symbol \diamond to convey the idea that the type to the left had more operations than the type to the right. Nowadays, this relation is usually written $<:$ and called *subtyping*, which seems to convey exactly the opposite intuition.

Having settled on a conformity-based type system, we still had to address the question of whether conformity should be deduced or declared. In other words, would it be sufficient for an object to *have* all of the operations demanded by a type, or would it also be necessary for the programmer to *say* that it had them? Because URLs would not be invented

for another 10 years, and even local-area distributed file systems were rare and primitive, there was no simple way for a programmer to state that one type (declared right here) conformed to another type (declared in some other program on some other computer). Also, our experience with Eden had taught us that we would often not appreciate the need for a “supertype” (a type with fewer operations) until after someone else had written a program using a subtype. It seemed quite impractical to have to ask some other programmer, possibly in some other organization, to change his or her code to say that one of the types that it used conformed to a supertype definition that we had written later. It also seemed pointless: it was easy enough to check type conformity directly.

This question of whether type compatibility should be deduced or declared is currently still open; the current jargon is to call deduced conformity *structural* and declared conformity *nominal*. Four or five years after we had chosen structural equivalence for Emerald, Cardelli and his colleagues wrote:

there is a strong argument for switching to structural equivalence, which is that structural equivalence makes sense between types that occur in different programs, while name equivalence makes sense only between types that occur in the same program. This advantage becomes significant when type-safety is extended to distributed systems... or to permanent data storage systems [39, p. 207].

Modula-3, a version of Modula designed for distributed systems, moved from the nominal typing of Modula-2 to structural typing [38].

We were not, of course, the first to use structural equivalence—Algol-68 and Euclid were there before us. Neither were we the first to realize that in an object-based language, “structure” meant not the layout of data fields, but the availability of operations—Smalltalk had done that, and its lead has since been followed by more recent languages such as Python and Ruby, which call it “duck typing”. (The name comes from the idea that if it looks like a duck, walks like duck, and quacks like a duck, it must *be* a duck [107].) But we were perhaps the first to apply static duck typing to objects.

4.2.4 Type Checking and Error Messages

Another consequence of the open world assumption was that sometimes type checking had to be performed at run time, for the very simple reason that neither the object to be invoked nor the code that created it existed until after the invoker was compiled. This requirement was familiar to us from our experience with the Eden Programming Language [21]. However, Eden used completely different type systems (and data models) for those objects that could be

created dynamically and those that were known at compile time.

For Emerald, we wanted to use a single consistent object model and type system. Herein lies an apparent contradiction. By definition, compile-time type checking is done at compile time, and an implementation of a typed language should be able to guarantee at compile time that no type errors will occur. However, there are situations where an application must insist on deferring type checking, typically because an object with which it wishes to communicate will not be available until run time.

Our solution to this dilemma provided for the consistent application of conformity checking at either compile time or run time. If enough was known about an object at compile time to guarantee that its type conformed to that required by its context, the compiler certified the usage to be type-correct. If not enough was known, the type-check was deferred to run time. In order to obtain useful diagnostics, we made the design decision that such a deferral would occur only if the programmer requested it explicitly, which was done using the `view...as` primitive, which was partially inspired by qualification in Simula 67 [15, 41].

Consider the example

```
var unknownFile: File  
...  
r ← (view unknownFile as Directory).Lookup["README"]
```

Without the `view...as` *Directory* clause, the compiler would have indicated a type error, because `unknownFile`, as a *File*, would not understand the *Lookup* operation. With the clause, the compiler treated `unknownFile` as a *Directory* object, which would understand *Lookup*. In consequence, `view ... as` required a dynamic check that the type of the object bound to `unknownFile` did indeed conform to *Directory*. Thus, successfully type-checking an Emerald program at compile time did not imply that no type errors would occur at run time; instead it guaranteed that any type errors that *did* occur at run time would do so at a place where the programmer had explicitly requested a dynamic type check.

The `view...as` primitive later appeared in C++.

Partially inspired by the `inspect` statement of Simula 67 [15, 41], we also introduced a Boolean operator that returned the result of a type check. This allowed a programmer to check for conformity before attempting a `view...as`.

4.2.5 Types and efficiency

A primary goal of Emerald was to demonstrate the viability of using a single object model for both small (*Integer*) and large (*Directory*) objects. One of our performance goals was to achieve the performance of C for simple operations like adding integers and invoking operations on local objects. We believed that static typing would lead to improved efficiency

and we used information from the type system in two places: primitive types and operation invocation.

Primitive types

We realized that a few primitive types must behave correctly for the language to be usable. In particular, consider the Boolean type. The correctness of the `if` statement and `while` loop depend on the proper behaviour of the two Boolean objects `true` and `false`⁵. We therefore insisted that a few types would not follow the normal rules for conformity: no non-primitive type conforms to Boolean. Eventually, we extended this notion for performance as well as correctness and defined a collection of primitive object types that included *Boolean*, *Character*, *Integer*, *Real*, *String*, and *Vector*. When the compiler knew that a variable had a primitive type, it also knew the implementation of the object bound to the variable, and used the direct object implementation shown in Figure 9. This meant that operations on such objects could be inlined and be made as efficient as in a conventional language.

Operation Invocation

A performance problem plaguing object systems that were contemporary with Emerald was the cost of finding the code to execute when an operation was invoked on an object. This process was then generally known by the name “method lookup”; indeed it still is, but we in the Emerald team called it operation invocation. In Smalltalk, method lookup involved searching method dictionaries starting at the class of the target object and continuing up the inheritance class hierarchy until the code was located. We thought that if Emerald didn’t do static type checking, each operation invocation would require searching for an implementation of an operation with the correct name, which would be expensive—although, because we did not provide inheritance, not as expensive as in Smalltalk. In a language like Simula in which each expression had a static type that uniquely identified its *implementation*, each legal message could be assigned a small integer and these integers could be used as indices into a table of pointers to the code of the various methods. In this way, Simula was able to use table lookup rather than search to find a method (and C++ still does so). We thought that static typing would give Emerald the same advantage, and this was one of the motivations for Emerald’s static type system.

However, even with static typing, there is still a problem in Emerald: except for the above-mentioned primitive types, knowing the type of an identifier at compile time tells us *nothing* about the implementation of the object to which it will be bound at run time. This is true even if the program submitted to the compiler contains only a single implementation that conforms to the declared type, because it is al-

⁵ Even in Smalltalk, in which conditional statements are represented by message sends, messages such as `ifTrue:ifFalse:` are known to the compiler and treated specially; it is not in practice feasible to re-implement Boolean.

ways possible for another implementation to arrive over the network from some other compiler. Thus, the Emerald implementation would still have to search for the appropriate method: the only advantage that static typing would give us would be a guarantee that such a method existed.

It is often the case that *dataflow analysis* can be used to ascertain that an object has a specific concrete type, and the Emerald compiler used dataflow analysis quite extensively to avoid method lookup altogether, by compiling a direct subroutine call to the appropriate method. However, the point that we did not fully appreciate when we started the Emerald project was that static typing, in itself, would *not* help us to avoid method lookup.

In those cases where dataflow analysis could not assign a unique concrete type to the target expression, we avoided the cost of searching for the correct method by inventing a data structure that took advantage of Emerald's abstract typing. This data structure was called an *AbCon*, because it mapped *Abstract* operations to *Concrete* implementations. The run-time system constructed an AbCon for each $\langle \text{type}, \text{implementation} \rangle$ pair that it encountered. An object reference consisted not of a single pointer, but of a pair of pointers: a pointer to the object itself, and a pointer to the appropriate AbCon, as shown in Figure 3.

The AbCon was basically a vector containing pointers to some of the operations in the concrete representation of the object. The number and order of the operations in the vector were determined by the abstract type of the variable; operations on the object that were not in the variable's abstract type could never be invoked, and so they did not need to be represented. In Figure 3, the abstract type *InputFile* supports just the two operations *Read* and *Seek*, so the vector is of size two, even though the concrete objects assigned to *f* might support many more operations. AbCon vectors were created where necessary when objects were assigned to identifiers of a different type, and were cached whenever possible to avoid recomputing them. AbCons increased the cost of each assignment slightly, but made operation invocation as efficient as using a virtual function table. In practice it was almost never necessary to generate them during an assignment, because the number of different concrete types that an expression would take on was limited, often to one. We compare AbCons with more recent technologies in Section 6.4.

4.2.6 Type:Type

As we mentioned in Section 3, the occasional need to defer type checking until run time implied that types would have to be representable at run time. Because Emerald was object-based, it seemed like an obviously good idea that types should themselves be represented as objects. The alternative would be to increase the size of the language dramatically by providing one set of declaration and parameteriza-

tion constructs for objects and another parallel set for types. Our minimality goal discouraged full exploration of this alternative. A consequence of this decision was that Emerald's type system would have the *Type:Type* property, that is, the property of an object being a type would be a type property, just like the property of being an integer or the property of being a set. At this time, the papers investigating *Type:Type* [36, 77] had not yet been published, and we didn't see *Type:Type* as a bad thing. Later, we realized that one of the consequences of *Type:Type* was that Emerald's type system was undecidable: there were certain pathological type checks involving infinite type objects that would not terminate. But this didn't seem to be a real problem either: these infinite type checks would occur only at run time, and the possibility of computations that did not terminate at run time had always been with us.

Once types were objects, the distinction between types and non-types was no longer one of syntax, but one of value. Thus, arbitrary expressions might appear in positions that required types. Such expressions were evaluated by the compiler, resulting in type objects, the *values* of which were used to do type checking. For example, in the declaration

```
var x : Integer
```

the expression *Integer* was evaluated, resulting in an object *v*. The type system then inspected *v* (i.e., it looked at *v*'s *value*) in order to assign a type to the identifier *x*. Clearly, the context implied that *v*'s value should be a type, in other words, that *v* \supseteq *Type*⁶; if it did not, the compiler signaled an error. Thus, we see that the values of certain objects, called type objects, were manipulated at compile time to do type checking. These same type objects were also available at run time to perform dynamic type checking.

For pragmatic reasons, the compiler restricted the expressions that could appear in a type position to those that were *manifest*. Intuitively, a manifest expression was one that the compiler could evaluate without fear of either non-terminating computations or (side) effects. We could guarantee a computation to be free from effects by insisting that only functions on immutable objects with immutable arguments that returned immutable results were evaluated at compile time. In addition, while it was obviously not decidable whether or not an arbitrary computation would diverge, the compiler placed restrictions on what it was willing to evaluate to guarantee that compilation terminated. It turned out that we never found a need for alternation (*if*) or iteration

⁶The identifier that we initially chose to denote the type of all types was *AbstractType*; we used the keyword **type** to signify the start of a type constructor, a special form of an object constructor that created a type object. Later, we reversed this decision: we used *Type* for the type of all types, and **typeobject** for the special object constructor. In this article we use the more recent syntax consistently; we felt that changing notation part way through the text, although historically accurate, would be unnecessarily confusing.

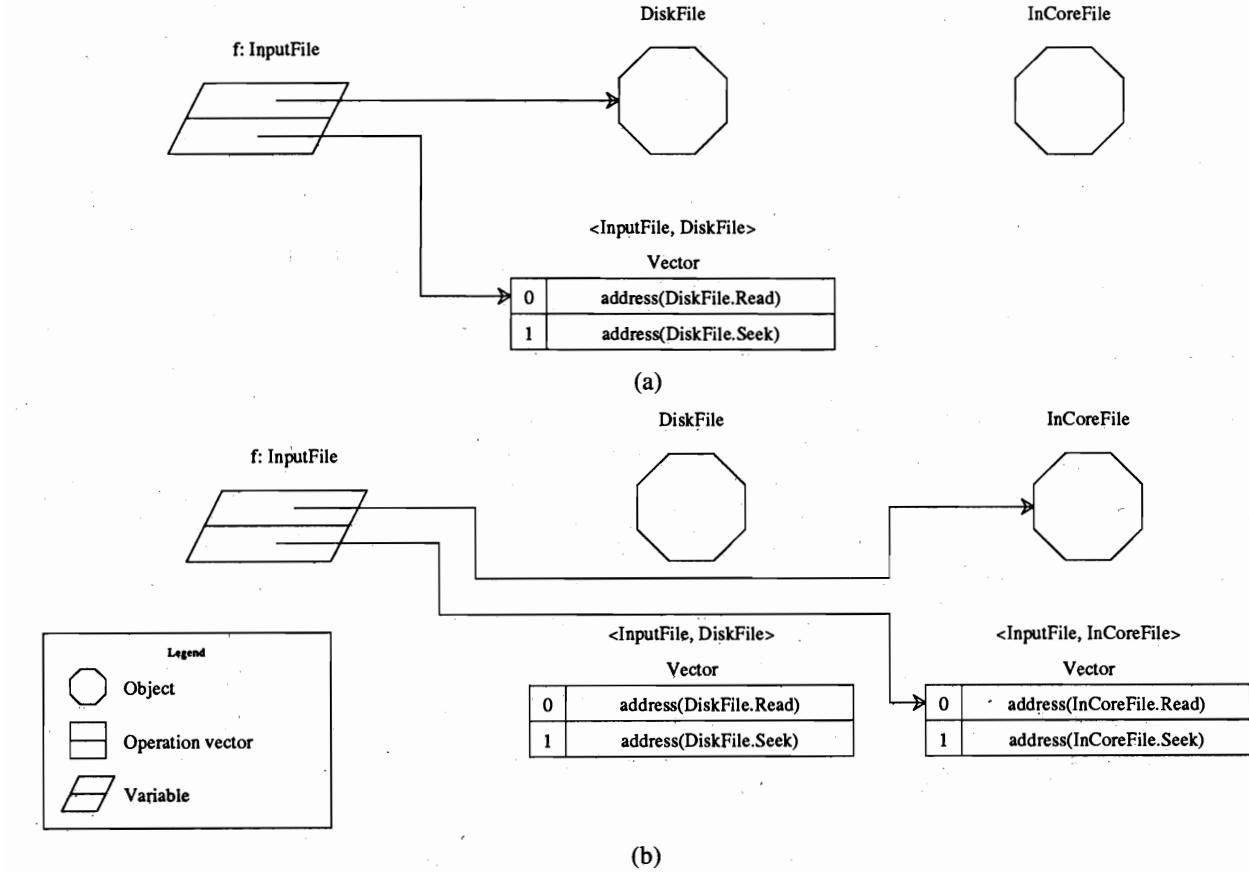


Figure 3: This figure, taken from reference [18], shows a variable *f* of abstract type *InputFile*. At the top of the figure (part a), *f* references an object of concrete type *DiskFile*, and *f*'s AbCon (called an Operation vector in the legend) is a two-element vector containing references to two *DiskFile* methods. At the bottom of the figure (part b), *f* references an object of concrete type *InCoreFile*, and *f*'s AbCon has been changed to a two-element vector that references the correspondingly named methods of *InCoreFile*. (Figure ©1987 IEEE; reproduced by permission.)

tion (**for**, **while**) in our manifest expressions. Thus, when we wrote

```
var v : Vector.of[Integer]
```

although *Vector* was just a constant identifier that happened to name a built-in object, and *of* was just an operation on that object, because the objects named by *Vector* and *Integer* were both immutable, and because *of* was a function, the evaluation of the expression *Vector.of[Integer]* could proceed at compile time; we knew that the value of this expression would be the same at compile time as it would be at run time.

Once types were first-class objects, and comparison for conformity (rather than equality) was the norm, we realized that

it would be a small step to allow an object that was a type also to have additional operations. For example, the object denoted by the predefined identifier *Time*, in addition to being a type, also had a *create* operation that made a new time from a pair of integers representing the number of seconds and μ s since the epoch. The Emerald programmer was thus able to define objects that acted as types and *also* had arbitrary additional operations; this was particularly useful for factory objects, which could be both the creators of new objects and their types.

4.2.7 Conformity, nil, and the Lattice of Data Types

The Nature of Conformity. The role of conformity in Emerald was so central that in our early discussions we

treated it as a relation between an object and a type. Other operations on objects could be defined in terms of conformity. For example, to ascertain whether or not an object o possessed an operation f with one argument and one result, we could evaluate:

```

$$o \diamond \text{typeobject } T
\quad \text{operation}^7 f[\text{None}^8] \rightarrow [\text{Any}]
\text{end } T$$

```

However, in working with Larry Carter in 1985-6 to formalize the definition of conformity, we realized that conformity needed to be a relation between *types*: the definition of conformity is recursive and depends on the conformity of parameters and results. Consequently, the paper that presents that definition [18] is somewhat inconsistent, referring in places to objects conforming to types, and elsewhere to types conforming to each other.

Unlike languages with which we were familiar, the notion of conformity meant that every object had not one but many types. Referring to Figure 4, any object that had type *ScansableDirectory* also had type *Directory*. *DeleteOnlyDirectory*, *AppendOnlyDirectory* and *Any*, among others. Thus, we found ourselves talking not about “the” type of an object but about its “best-fitting type”, meaning the type that captured all of the operations understood by the object. We realized that \diamond induced a partial order on types, as depicted in Figure 4. Because in Emerald the type *Any* had no operations, it was the least element in this partial order. Some types were incomparable: the type with *Add* as its only operation seemed to be incomparable to the type with only *Delete*. Nevertheless, these two types had *Any* as their greatest lower bound.

The Partial Order of Types. Although this partial order was for the most part simple and intuitive, we were aware of two problems with it. The first problem, which confused us for a long time, arose when two types had operations with the same name but with different arities, i.e., different numbers of arguments or results. For example, consider two types, one with the operation $\text{Add } [\text{String}] \rightarrow []$ (*Add* with a single *String* argument and no result), and one with the operation $\text{Add } [\text{Integer}] \rightarrow [\text{Integer}]$ (*Add* with a single argument and a single result, both *Integer*). Not only are these types incomparable, but they seemed to have no upper bound: there was no type to which they both conformed, because no type could have an single *Add* operation with both of these arities.

A second problem was how to type **nil**, the “undefined” object. We typically wish to use **nil** in assignments:

```
var d : Directory
d  $\leftarrow$  nil
```

⁷ The syntax **operation** $\text{name}[T] \rightarrow [U]$ means that the method *name* takes one argument, which is of type *T*, and returns one result, of type *U*.

⁸ *None* is the type that includes every possible operation, and *Any* is the type that includes no operations, as described later in this Section.

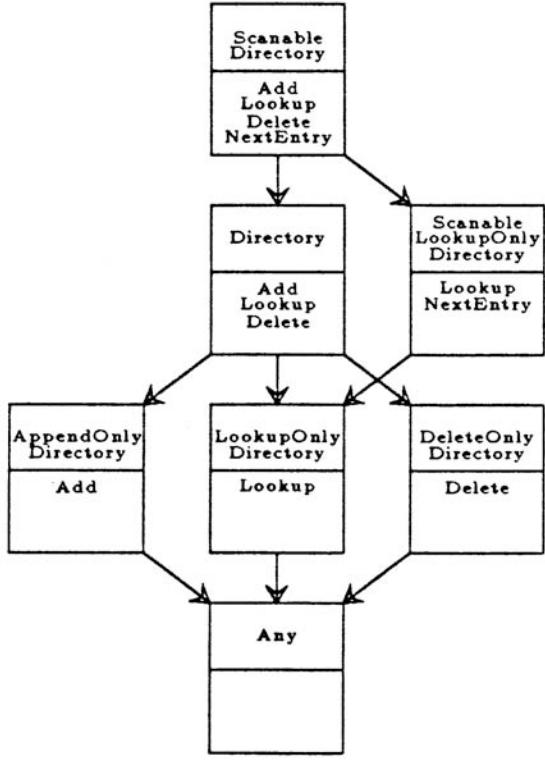


Figure 4: An example of a directed acyclic graph (July 1988) that illustrates the partial order on types induced by conformity. Each box represents a type; above the line is the name of the type, and below is a list of the type’s operations. The arrows represent the conformity relation, for example, *ScansableDirectory* \diamond *Directory* because it has a superset of *Directory*’s operations.

and in tests:

```
if d  $\neq$  nil then r  $\leftarrow$  d.Lookup[“key”] end if
```

It should be clear that for **nil** to be assignable to *d*, it must support all the *Directory* operations. Similarly, for **nil** to be assignable to **var** *i* : *Integer* it must support all of the *Integer* operations. By extension, **nil** must possess all of the operations of all possible types that might ever be constructed! This seemed like a contradiction, because operationally we knew that **nil**, far from being an all-powerful object, actually did nothing at all.

The conventional solution to this dilemma, and the one that we initially adopted for Emerald, was to make **nil** a special case. Either **nil** would refer to a single special object that did not otherwise fit into the type system, or, as in Algol 68, there would be a separate **nil** for each reference type, and a syntactic mechanism for disambiguating the overloaded symbol **nil** that denoted them all [103, §2.1 and §3.2].

From Partial Order to Lattice. We eventually stumbled on an elegant solution to both the arity problem and the nil problem. In the partial order so far described, every pair of

types T and U had a “meet” (also known as greatest lower bound) $T \sqcap U$ that contained just those operations that were common to T and U . If T and U had no operations in common their meet was *Any*, so *Any* was the least element in the partial order. If T and U had in common just operation $\alpha [f] \rightarrow [g]$ then the type containing just α with that signature was their meet $T \sqcap U$. However, because of the arity problem, we could not see how to define the “join” (least upper bound) of arbitrary types. When an operation α had different arities in types T and U , the meet of T and U was well-defined (it would omit α completely), but it seemed that the join did not exist.

Meets and joins arose rather naturally when performing type checking. For example, if a variable could take on either a value of type T or a different value of type U , all that we could say about the type of that variable is that it is the meet of T and U . Because of contravariance, computing this meet might involve computing a join, for example, if T and U both supported operations α with the same arity but with different signatures, say, $\alpha [f_T] \rightarrow [g_T]$ and $\alpha [f_U] \rightarrow [g_U]$, then $T \sqcap U$ was the type containing $\alpha [f_T \sqcup f_U] \rightarrow [g_T \sqcap g_U]$ —provided that $f_T \sqcup f_U$ (*the join of the f_τ*) was defined. This definition generalized in the obvious way to types with more than one operation, and to operations with multiple arguments and results.

The problem with this definition was the inelegant caveat that the required meets and joins must all exist. Because of contravariance, once some joins did not exist, it was also possible for some meets not to exist: the problem cascaded. Black had studied at Oxford under Stoy and Scott, and knew that the conventional mathematical solution to this problem was to ensure that *all* upper and lower bounds exist, i.e., to embed the partial order in a complete lattice. He also knew that we would not lose any generality by this embedding, because such a lattice always exists [98, pp. 88–91, 414]. However, for a long time we could not see how to construct it, because we could not see what to do about an operation that had different arities in T and U .

The solution, like many good ideas, is quite simple and in hindsight quite obvious: we needed to treat operations with different arities as if they had different names. For practical purposes this meant that we changed the language to allow overloading by arity; formally, we equipped every operation name with two subscripts, representing the number of arguments and results. This meant, for example, that the two add operations mentioned above became $Add_{1,0}$ and $Add_{1,1}$. Now that the operations had distinct names, the upper bound could contain both of them, and as a consequence we were able to turn the partial order into a lattice.

***None*.** Of course, every lattice has a unique top element; what surprised us initially was the discovery that the top element of Emerald’s type lattice was semantically useful. We realized that it provided a type for **nil**, so we called it

None; this also seemed like an appropriate name for the dual of *Any*.

We defined $T \sqcap U$ as the largest type (that is, the type with the largest number of operations) such that $T \diamondsuit (T \sqcap U)$ and $U \diamondsuit (T \sqcap U)$, and defined $T \sqcup U$ as the smallest type such that $(T \sqcup U) \diamondsuit U$ and $(T \sqcup U) \diamondsuit T$. The type of all objects, which we called *Any*, was then nothing more than the bottom element of the lattice induced by the conformity relation: *Any* was the maximal type such that for all types T , $T \diamondsuit \text{Any}$. Dually, we defined *None* to be the top of this lattice: *None* was the minimal type such that for all types T , $\text{None} \diamondsuit T$. The keyword **nil** then simply denoted the (unique) object of type *None*. By this definition, **nil** supported all possible operations, with all possible signatures. In other words, **nil** supported the operation *Add* with 1, 3, and 17 arguments of the most permissive types, as well as the operation *Halt* [*Turing-machine, Tape*] \rightarrow [*Boolean*]. This didn’t initially sound like the **nil** with which we were familiar! However, we finally realized that the conformity lattice spoke only about type checking. If any of these all-powerful operations were actually *invoked*, the implementation would immediately break—which is exactly the behavior we expected of **nil**.

Restriction and Capabilities. In Eden, objects were addressed using a capabilities that included not only a reference to the object but also a set of access rights that described which operations were invocable using the capability. Applications could create capabilities with restricted access rights and send them to clients. For example, the creator of a mail message object would have the right to perform all operations on it, but might send the recipient’s restricted capabilities that gave them the right to read the message but not to modify it. The access rights were implemented as a fixed-length bit vector, which provided a small name space and did not mesh well with subtyping [22].

Emerald’s type system provided a similar facility, in that clients of an object could be given a reference with a restricted type. However, the **view...as** facility meant that the type restriction could always be circumvented. Apart from this fatal flaw, the type-based mechanism was better than Eden’s access rights: it resolved the small-name-space problem, and it was compatible with subtyping.

During the summer of 1987, Norm visited Seattle and met Eric at a Lake Washington café. Over breakfast they designed a new language primitive **restrict...to**, which removed the flaw by limiting the power of the **view** facility. To see how **restrict** works, consider the code in Figure 5, which references a *Directory* object using the types of Figure 4. In the figure, a single object conforming to *Directory* is bound to d , l , and lr . Both l and lr have type *LookupOnlyDirectory*, but whereas l ’s reference is unrestricted, and can be “lifted” back up to *Directory* by a **view** expression, lr ’s reference is restricted to *LookupOnlyDirectory*. Thus, the **view** expres-

```

var d, d1, d2 : Directory
var lr : LookupOnlyDirectory
d ← ...
l ← d
lr ← restrict d to LookupOnlyDirectory
d1 ← view lr as Directory
d2 ← view lr as Directory

```

Figure 5: The **restrict...to** primitive can be used to limit the operations invokable on an object. The types *Directory* and *LookupOnlyDirectory* are shown in Figure 4.

sion in the assignment to *d1* will succeed, whereas the view expression in the assignment to *d2* will fail. This is because the restricted reference stored in *lr* cannot be viewed above the level of the *LookupOnlyDirectory* in the type lattice.

The restrict mechanism was implemented by late afternoon of the day on which it was designed. The implementation was simple. AbCons already contained a reference to the concrete type of the object; all that was necessary was to introduce an extra field that specified the largest type allowed in a **view...as** expression. Upon creation of an Ab-Con this field was initialized to the concrete type, but a **restrict...to R** expression would create an AbCon containing the restricted type *R*.

Overloading. In spite of this rather theoretical genesis, our decision to include overloading by arity but not by type was made because it worked well for the practicing programmer. We now believe that overloading by arity represents a “sweet spot” in the space of possibilities for overloading. It supports the most common uses, such as allowing unary and binary versions of the same operation, and allowing the programmer to provide default values for parameters. Nevertheless, it is always easy for the programmer (and the compiler) to know which version of the operation is being used. In contrast, overloading by type, as provided in Ada at the time of Emerald’s development and as now adopted in Java, is simply a bad idea in an object-oriented language. Many programmers confuse overloading with dynamic dispatch; it is difficult for both the programmer and the compiler to disambiguate; and it does little more than encourage the programmer to be lazy in inventing good names, for example, by permitting both *moveTo(aPoint)* and *moveBy(aVector)* to be called *move*. In a procedural or functional language, it is hard to argue that procedures like *print* and functions like *=* and *+* should not be overloaded. But in an object-based language, these things are just operations on objects, and several operations with the same name can be implemented by different objects without any need for type-based overloading.

Implementing nil. There was no difficulty in implementing **nil** for objects that were represented by pointers: we just chose a particular invalid address for **nil** whereby we had a free hardware check for invocations of **nil**. However, the attentive reader will recall from Section 4.1.4 that reals and in-

tegers were primitive and were represented directly, without using pointers. We wanted to avoid a software-based (and thus highly inefficient) check for **nil** on integer and real operations. We couldn’t find a perfectly clean representation for **nil** that was also efficient, and in the end in our implementation of Emerald we cheated just a little. We reserved a special **nil** value that could be assigned to real and integer variables. The bit pattern 0x80000000 was illegal as a VAX floating-point number, so by using this value to represent **nil** we could be sure that a legitimate floating-point operation would never result in **nil**. Moreover, the hardware would trap if this bit pattern were used in a floating-point instruction, so it was easy to detect an attempt to invoke an operation on **nil**. For integers, the same bit pattern represented -2^{31} , so by using this value for **nil** we were “stealing” the most negative value from the range of valid integers. If the program asked whether or not an integer variable was **nil**, the answer would be right. However, we did not implement checks to ensure that invocations on integer **nil** always failed or that normal arithmetic operations didn’t generate **nil** by accident. We were not happy with the idea that multiplying -2^{30} by 2 would evaluate to **nil**; neither were we happy that multiplying -2^{30} by 3 would give the wrong answer. The decision not to implement checks for integer overflow and underflow was entirely a matter of efficiency: we wanted built-in integers to be as efficient as C’s integers, and C didn’t do any checks, so we didn’t do any either. Of course, if a programmer wanted a integral numeric type that did all of the checking, or had a greater range, or whatever, such a integer could be implemented as an Emerald object—with the consequent performance penalty. But the built-in *Integer* type was quick, and in this case just a little bit dirty.

4.2.8 Polymorphism

In 1985, we felt that any “modern” programming language should have a type system that provided support for polymorphic operations, that is, operations that might be invoked in several different places with an argument of different types. The contemporary authority on types, a survey paper by Cardelli and Wegner [40], distinguished two broad classes of polymorphism, ad hoc and universal. We were not particularly concerned with ad hoc polymorphism; of universal polymorphism they wrote:

Universally polymorphic functions work with an infinite number of types, so long as these types share a common structure. As suggested by the name, only universal polymorphism is considered *true* polymorphism. Within this broad class are two sub-divisions:

inclusion polymorphism

An object can be viewed as belonging to many different types that need not be disjoint.

parametric polymorphism

A function has an implicit or explicit type param-

eter which determines the type of the argument for each application of the function.

We found these definitions confusing because the distinction between parametric polymorphism and inclusion polymorphism appeared to break down in Emerald. Consider the passing of an argument o of type S to an operation that expects an argument of type T such that $S \diamond> T$. This could be said to be an example of inclusion polymorphism because the object o has a number of different types including S and T (as well as *Any* and a host of others). An alternative explanation is that, because o knows its own type, the type of o is an implicit argument to every operation to which o is passed, making this an example of parametric polymorphism.

We finally realized that the distinction was still a valid one for Emerald: it lay in the way that the operation *used* its argument. If the operation treated o as having a fixed type T , then the inclusion polymorphism view was appropriate, but if it treated o parametrically, then the parametric view was appropriate (See the typechecking technical report [26] for some examples.)

As a consequence, we included two features in the Emerald type system, one for each of these two kinds of polymorphism. First, basing type checking on conformity rather than equality supported inclusion polymorphism: every operation that expected an argument of type T would also operate correctly on an argument whose type conformed to T .

Second, Emerald types were values (more precisely, objects), and we allowed arbitrary expressions to appear in syntactic positions that required types. This implied that types could be passed as parameters to operations. An operation could also return type as its result, and that type could then be used in a declaration. Figure 6 is an example of the sort of parametric polymorphism that we provided in Emerald: a polymorphic pair factory that insists that both elements of the pair are of the same type. A trivial application of this object to create a pair of strings might appear as follows.

```
const p ← pair.of[String].create["Hello", "World"]
```

The *of* function of *pair* accepts any type as its argument, but returns a pair that has no operations other than *first* and *second*. Suppose that we want the pair to also have an equality operation $=$ that tests the component-wise equality of one pair against another. Obviously, such a pair cannot be constructed for arbitrary element types: the element type must itself support an $=$ operation. We therefore need to be able to express a constraint on the argument to the *of* operation on *pair*. This can be done by adding a constraint on the *value* of the argument T passed to *of*. The constraint that we need is that:

```
T ⓥ> typeobject eq
  function = [eq] → [Boolean]
end eq
```

```
const pair ← immutable object pPair
  export function of[T : Type] → [r : PairCreator]
    where
      PairCreator ← typeobject PC
        operation create[T, T] → [Pair]
      end PC
    where
      Pair ← typeobject P
        function first → [T]
        function second → [T]
      end P
    forall T9
      r ← immutable object aPairCreator
        export operation create[x : T, y : T] → [r : Pair]
          r ← object thisPair
            export function first → [r : T]
              r ← x
            end first
            export function second → [r : T]
              r ← y
            end second
          end thisPair
        end create
      end aPairCreator
    end of
  end pPair
```

Figure 6: A polymorphic pair factory (after an example in a November 1988 working paper).

Syntactically, we gave Emerald a **suchthat** clause to capture this constraint. Types constrained in this way provide what Canning and colleagues later called *F-bounded polymorphism* [34] because the bound on the type T is expressed as a function of T itself. The **where** clause of CLU [73] provided similar power, although we were unaware of this at the time. We eventually realized that the $\diamond>$ relation in the **suchthat** did not denote the conformity relation that we had defined between types, but was actually a higher-order operation on *type generators*. In the types technical report [26] we called this operation matches, and denoted it by the symbol \triangleright . Subsequently, Kim Bruce adopted an equivalent definition for an operation that he denoted by \leq_{meth} [32], and eventually also called matches [33].

Two other changes are necessary in Figure 6 to define a pair with equality. The first is, obviously, the addition of the $=$ operation to the type *Pair*; the second is the addition of a corresponding implementation of $=$ to the object *thisPair*. The complete factory for pairs with equality is shown in Figure 7.

⁹The **forall** keyword was added quite late in Emerald's development. In a sense the **forall** is unnecessary, because it expresses the empty constraint on T . However, without it there is no declaration for the identifier T , and we felt that *every* identifier should be declared explicitly; the alternative of making the programmer write **suchthat** $T \diamond> Type$ seemed overly pedantic.

```

const pair ← immutable object pPair
export function of[T : Type] → [r : PairCreator]
where
  PairCreator ← typeobject PC
    operation create[T, T] → [Pair]
  end PC
where
  Pair ← typeobject P
    function first → [T]
    function second → [T]
    function = [P] → [Boolean]
  end P
suchthat T ⊢ typeobject eq
  function = [eq] → [Boolean]
end eq
r ← immutable object aPairCreator
export operation create[x : T, y : T] → [r : Pair]
  r ← object thisPair
    export function first → [r : Pair]
      r ← x
    end first
    export function second → [r : Pair]
      r ← y
    end second
    export function = [other : Pair] → [r : Boolean]
      r ← other.first = x and other.second = y
    end =
  end thisPair
  end create
end aPairCreator
end of
end pPair

```

Figure 7: A Pair Factory that is polymorphic over types that support equality

4.2.9 Publications on Types

Our initial ideas on types were published in an August 1985 technical report [19]. However, that report did not contain a formal definition of the conformity relation, not because we thought it unimportant, but because we were not sure how to do it. The obvious definition of conformity was as a recursive function, and we did not know how to ensure that the recursion was well-founded. Fortunately, Larry Carter from IBM was on sabbatical at UW in the fall of 1985, and we were able to convince him to help us formulate the definition. Larry’s definition, which constructed the conformity relation as the limit of a chain of approximations, appeared in a revised version of the technical report [16], and was eventually published in *Transactions on Software Engineering* in January 1987 [18].

As we deepened our understanding of the issues around types, and in particular looked harder at polymorphism, we revised a number of the decisions made in these early pa-

pers and developed more formal underpinnings for the type system. We also redefined conformity using inference rules, a technique that was then becoming popular. Andrew and Norm were responsible for this evolution and for authoring several documents describing it, none of which was formally published. Our earliest document is a handwritten paper “The Lattice of Data Types”, dated 1986.11.04. Sometime in 1987 this paper acquired a subtitle “Much Ado About NIL”; it eventually evolved into “Types and Polymorphism in the Emerald Programming Language”, which went through a series of versions between July 1988 and July 1989, before finally morphing into a technical report “Typechecking Polymorphism in Emerald”, which Andrew and Norm finished over the Christmas holidays in 1990 [26].

4.3 Concurrency

We knew that concurrency was inherent in any distributed system that made use of multiple autonomous computers. Moreover, we realized that even in a non-distributed program, Emerald’s object model implied that each object was independent of all others and was capable of acting on its own. This meant that every object should be given the possibility of independent execution; the consequence was that we needed a concurrency model that allowed concurrency on a much finer grain than that provided by operating system processes in separate address spaces.

4.3.1 The Emerald Process Model

In designing Emerald’s process and concurrency control facilities, we were inspired by Concurrent Pascal [29], with which Eric had worked extensively while writing his Master’s thesis [57] at the University of Copenhagen, and Concurrent Euclid [50, 51], which we had used in Eden. Note that, as we explained on page 4, we use the unqualified term *process* to mean what it meant at the time: one of a number of lightweight processes sharing an operating system address space. Today this would more likely be called a *thread*; when we mean an operating system process (in a protected address space), we say so explicitly.

In Eden, each object was a full-blown UNIX process and thus had its own address space, within which the Eden Programming Language provided lightweight processes. The two levels of scheduling and the costs of operating system interprocess communication inherent in this scheme were part of the cause of Eden’s poor performance. So, for Emerald, we knew that we needed to find a way to develop a single process model for all concurrency. We also knew that we would have to implement it ourselves, rather than delegating that task to an underlying operating system.

Our first idea was to have a separate language construct to define a process, modeled on the **process** of Concurrent Pascal. However, we soon realized that this would not be adequate. The Concurrent Pascal construct is static: all the processes that can ever exist must be defined at compile

time. This restriction could be lifted by making the creation of processes dynamic, for example, by allowing a **new process** construct similar to **new** in Simula. However, we were not happy about the idea of introducing another first class-citizen besides objects.

Object constructors had let us avoid introducing classes as first-class citizens: instead we nested one object constructor inside another. We realized that we could also nest processes *inside* objects. We added an optional **process** section to object constructors so that, when an object was created, a process could be created as one of the object's components. The process would start execution as soon as the object had been initialized. If the process section were omitted, the object would be passive and would execute only in response to incoming invocations. This design allowed us to have active objects that could execute on their own, as well as conventional passive objects. It also had the benefit of making it clear where a program should start executing. Some languages express this with various kinds of ad-hocery, such as a “special” process called “main”. Emerald needed no “special” process. Instead, we saw the Emerald world as a vast sea in which objects floated. When a program created a new object (by executing an object constructor), the object was merely added to the sea. If the new object had no process, nothing more happened (until the new object was invoked). If it did have a process, the process started execution concurrently with all the other Emerald processes.

Thus, instead of needing a “main procedure”, an Emerald program was simply a collection of object declarations. When the program was loaded, these declarations were elaborated, and any objects thus created were added to the sea. A sequential Emerald *Hello, world!* program looked like this.

```
const simpleprogram ← object myMainProgram
  const i ← 1
  process
    stdout.PutString[i.asString || “: hello, world!\n”]
  end process
end myMainProgram
```

This program uses a simple object constructor to create an essentially empty object whose only purpose is to house the process that prints “1: hello, world”, and then terminates.

We also considered allowing more than one **process** section in an object. This would have been of limited use: as a static construct, it would not have helped the programmer to create a variable number of processes in the object. The more general case of dynamically created processes was already provided: any number of processes could be created simply by defining internal objects that contained not much more than a process, and then creating such objects. This is illustrated in the example below.

```
const p ← object multiProgram
  export operation workerBody[i: Integer]
    stdout.PutString[i.asString || “: hello, world!\n”]
  end workerBody
  process
    var i: Integer ← 17
    var x: Any
    loop
      if i <= 0 then exit end if
      x ← object worker
        process
          multiProgram.workerBody[i]
        end process
      end worker
      i ← i - 1
    end loop
  end process
end multiProgram
```

This program contains a single object that creates 17 other *worker* objects each housing a process. These 17 processes thereafter execute in parallel and will print a message in some (indeterminate) order.

The following example shows a generalized worker process that is parameterized by a function containing the actual work to be done.

```
const WorkToDoType ← typeobject wtd
  operation doWork[ ]
end wtd

const workerCreator ← object wc
  export operation createWorker[work: WorkToDoType]
    var x: Any
    x ← object worker
      process
        work.doWork[ ]
      end process
    end worker
  end createWorker
end wc

const exampleWork ← object hello
  var i: Integer ← 0
  export operation doWork[ ]
    i ← i + 1
    stdout.print[i.asString || “: hello world!\n”]
  end doWork
end hello

const exampleProgram ← object exampleProgram
  process
    workerCreator.createWorker[exampleWork]
    workerCreator.createWorker[exampleWork]
  end process
end exampleProgram
```

The *workerCreator* operation is used to generate a process that executes the *doWork* operation of any object given to it as argument. In this example, the *hello* objects merely print messages. The object *exampleProgram* creates two worker

processes; the order in which the messages are printed is undefined because the two processes execute concurrently.

4.3.2 Synchronization

In general, many processes could be executing inside an object simultaneously. However, if they updated the same variables, race conditions could readily occur. Traditionally, such problems were resolved by protecting the shared variables with some form of synchronization construct. The final example in Section 4.3.1 contains a serious race condition in the object *hello* over the update of the variable *i* in the *doWork* operation. Because it was easy to write programs with unwanted race conditions, we provided a way for the programmer to state that an object's variables would be protected inside a monitor; the externally visible operations on that object would then become the monitor entry operations.¹⁰ Thus the *hello* object in the example becomes:

```
const exampleWork ← monitor object hello
  var i: Integer ← 0
  export operation doWork[ ]
    i ← i + 1
    stdout.PutString[i.asString || ": hello world!\n"]
  end doWork
end hello
```

We did not spend a lot of time considering alternatives to the monitor. Innovation in concurrency control was not a goal; we were familiar with monitors, they were known to be adequate, and they were widely taught and understood (although some of Black's prior work [1] indicated that they were not as well understood as we had thought!) We adopted Hoare semantics [49] for monitors, including the facilities for signaling and waiting on so-called condition variables. However, condition variables themselves posed a bit of a problem. In Hoare's original design, condition variables could be declared only inside a monitor, and consequently had a scope that was limited to the enclosing monitor. It was impossible to export a reference to a condition variable, and it made no sense to wait on a condition variable in one monitor and signal it in another. So conditions seemed to be much less general than objects, which were always known by reference and which could be passed around freely.

We initially followed Hoare's approach and defined a special system object, *Condition*, that returned a unique condition variable when its *create* operation was invoked. However, in line with our minimality goal (Section 2), we really wanted to avoid introducing any kind of special variable into Emerald. We realized that condition variables were not really variables at all in the conventional sense: they did not refer to values. The purpose of a condition variable was merely to

¹⁰We initially allowed any object to contain a monitored section. We later decided to instead make a whole object monitored; this simplified both the language and the compiler. The effect of an internal monitor could still be obtained with a nested object. For the sake of consistency, all of the examples in this paper use the current (whole object as monitor) syntax.

serve as a label for the logical condition for which a process might need to wait. Consequently, we realized that any kind of unique label would do. To avoid creating another concept, that of labels, we decided to use an arbitrary object as a label; after all, every object had a unique identity. Thus, in **signal** and **wait** statements, we allowed any object to be used to label the condition.

However, this simplification gave rise to another problem. If condition variables were not "special", how were we to enforce the restriction that a particular condition could be used within only a single monitor? Although this restriction would naturally follow from the practice of declaring the condition object locally within the monitor, this practice did not amount to a guarantee. Because condition objects were now just general-purpose objects, they could be stored inside other objects, passed as arguments, and so on: it would be impossible to limit their scope statically. We therefore decided to enforce the "single monitor" restriction dynamically. The implementation of condition variables created the structure that implemented the condition the first time that either **signal** or **wait** was applied to an object, and associated the condition structure with the enclosing monitor at that time. Subsequent applications of **signal** or **wait** checked that their condition argument had been associated with the same monitor. Because **signal** and **wait** were language statements, not operation invocations, it was trivial to ensure that they appeared only inside a monitor.

One additional advantage of using the identity of the object and not its representation was that this avoided any complications concerning distribution: nonresident objects still had resident object descriptors, so **signal** and **wait** never needed to access nonlocal data structures.

To help programmers express their intent, we retained the special system object *Condition*. A *Condition* object was an empty object without operations: *Conditions* were never invoked, but were merely used for their unique identity in **signal** and **wait** statements. In this way, we introduced the concept of a first-class label without making the language larger: labels were simply objects.

The code in Figure 8 shows two processes keeping in step with one another by alternating their execution. Each process executes an operation 10 times. They synchronize through a monitor, so that if one gets ahead of the other, it will have to wait its turn. The *Condition* object *c* represents the condition "it's my turn now"; the operations *Hi* and *Ho* **wait** on and **signal** *c*, but *c* is never invoked.

Monitors did not present any significant challenge related to mobility. The implementation structure for each monitor, including the monitor lock and any conditions, was packed up and moved along with its enclosing object. Processes that were waiting for entry into the monitor, either because they had invoked a monitored operation or because they

```

const initialObject ← object initialObject
const limit ← 10

const newobj ← monitor object innerObject
var flip : Boolean ← true % true => print hi next
const c : Condition ← Condition.create

export operation Hi
  if ! flip then
    wait c
  end if
  stdout.PutString[“Hi\n”]
  flip ← false
  signal c
end hi

export operation Ho
  if flip then
    wait c
  end if
  stdout.PutString[“Ho\n”]
  flip ← true
  signal c
end ho

initially
  stdout.PutString[“Starting Hi Ho program\n”]
end initially

end innerObject

const hoer ← object hoer
process
  var i : Integer ← 0
  loop
    exit when i = limit
    newobj.Hi
    i ← i + 1
  end loop
end process
end hoer

process
  var i : Integer ← 0
  loop
    exit when i = limit
    newobj.Ho
    i ← i + 1
  end loop
end process
end initialObject

```

Figure 8: One of the processes in the object *hoer* invokes the *Hi* operation on *newobj* 10 times; the other invokes *Ho*. Because these operations execute inside a monitored object, they operate in mutual exclusion and the output is an alternating stream of *Hi* and *Ho* messages.

were waiting on a condition, were moved just like any other process that was executing (or waiting) within an object.

4.4 Initially

In many languages, initializing variables and data structures was a bothersome task. In Emerald, the problem was further

compounded by concurrency: once created, a process ran in parallel with its creator. Consequently, race conditions could occur when creating a new object. For example, if a process *P* created a new object *A*, and during its creation *A* caused another process *Q* to be created, then *Q* might “outrun” *P* and try to invoke the new object *A* before *P* had finished initializing *A*. An object did not even need to create another process for a race condition to occur: if a new object *A* registered itself in a directory so that others could find it, then an aggressive process that noticed *A* in the directory might try to invoke *A* before *A* had finished its initialization. The same problem exists today in Java.

We solved this problem by locking an object until its **initially** section had completed. This enabled the body of the **initially** to use other objects freely, but a cycle would result in deadlock and would thus be easy to detect.

4.5 Finalization

Some object-based languages allowed the programmer to define so-called finalizers, also known as destructors. The idea was that just before an object was destroyed, its finalizer would be given a chance to “clean up”, for example, to close open files or to release allocated data structures. In our minds, objects lived forever, so a finalizer did not make sense. The garbage collector could recycle objects that were no longer of any use—which meant that they were not accessible from a basic root or by an executing process. We did consider introducing a finalizer that would be invoked in this situation, but once something was executing inside the object, it would no longer be a candidate for garbage collection. So finalizers would have violated an important monotonicity property: once an object became garbage, it would stay garbage.

4.6 Compiler-Kernel Integration

The Emerald compiler and run-time kernel were very tightly integrated (see Section 3). This was essential for accomplishing our performance goal. Tight integration allowed the compiler several forms of flexibility: it could select between the three object implementations (global, local, and direct, described in Section 4.1.4) for every object reference; it could use the general purpose registers to hold whatever data it liked; and it understood the format of kernel-maintained data structures and could inspect them directly, rather than calling a kernel primitive to interpret them.

The compiler was responsible for informing the kernel about its representation choices, and because the kernel could take control at (almost) any point in the execution and might need to marshal object data, the run-time stack, and even the processor registers, the compiler had to provide descriptions of every accessible data area at all times. These descriptions, called *templates*, described the contents of an area of memory. They informed the run-time system where immediate data (direct objects), object pointers (local object

references), and object descriptor references (remote object references) were stored. A particular template described either an object’s data fields or the contents of the processor registers and stack for a single activation record. Because the stack contents varied during the lifetime of an activation record we considered dynamic templates that would be a function of the program counter, but we avoided this complexity by ensuring that the variable part of the stack *always* contained full Emerald variable references, including AbCon pointers. The same templates provided layout information for the garbage collector, the debugger and, not the least, the object serializer. The object serializer was capable of marshaling *any* object, including those containing processes and active invocations, using the compiler-generated templates. This meant that, unlike other RPC-like systems (e.g., Birrell and Nelsons’ RPC [14]) there was no need for the programmer to be involved in serializing objects. Completely automatic serialization was also necessary because we did not give programmers access to low-level representations.

Fast Path and Object Faulting Allowing compiled code to inspect kernel data structures was key to making local invocations of global objects as fast as procedure calls in C. The implementation of an invocation followed either a “fast path” or a “slow path”. The fast-path invocation code sequence generated by the compiler checked a *frozen* bit in the object descriptor. If the object was not frozen, then the fast path code was free to construct the new activation record and jump to the appropriate method code. However, if the object was frozen, then the compiler-generated code took the slow path by calling into the kernel to perform the invocation. There were a large number of situations in which an object was frozen: it might have been still under construction, it might not have been resident on the machine, it might have failed, it might have been in the process of being moved, or it might need to be scanned by the garbage collector. The compiler-generated invocation sequence needed to identify whether or not it could use the fast path: all of the slow paths started with a call into the kernel that then ascertained which of the various special cases applied. We called this mechanism *object faulting* because it was similar to page faulting on writes to read-only pages. Even without hardware support the object-faulting mechanism was quite efficient and was crucial for the implementation of the parallel, on-the-fly, faulting garbage collector (seen Section 5.3).

4.7 Mobility

Although we could not know it at the time, the major advance of Emerald over most of its successors was that Emerald objects were mobile. Given our experience with Eden, mobile objects were the obvious way of meeting our goal of explicit support for object location.

Although operation *invocation* was location independent in Emerald, it was never a goal that *objects* should be location

independent. Indeed, we recognized that some objects, particularly those that needed to exploit hardware, would need to be placed on particular machines. We also thought that automating the placement of objects in a distributed system was in general too hard; instead we felt that it was the responsibility of the application programmer to place objects appropriately, given his or her knowledge of the application domain. We therefore gave Emerald a small number of location-dependent primitives, extending what we had done in Eden.

4.7.1 Location Primitives

Location was expressed using *Node* objects, each of which was an abstraction of a physical machine. For example, if *Y* were a node object, the statement **fix X at Y** locked the object *X* at location *Y*. However, *Y* was not restricted to be a *Node*; any object could be used as a location. So, if *X* was a mail message and *Y* a mail box, **fix X at Y** was still valid, and meant that *X* should be locked at the current location of *Y*. Andrew had first thought of the idea of using arbitrary objects to represent locations when designing Eden’s location-dependent operations; the idea had worked well, so we adopted it for Emerald.

Emerald had five location-dependent operations and two special parameter passing modes that influenced location. The location dependent operations were as follows.

- **locate** an object; the answer was a *Node* object that indicated where the target resided. There was no guarantee that the object might not move immediately afterwards.
- **move** an object **to** another location.
- **fix** an object **at** a particular node; this might have involved moving it there first. An attempt to fix an object would fail (visibly) if, for example, the target object were already fixed somewhere else. An **isfixed** predicate was also provided.
- **unfix** an object: make it movable again after a **fix**.
- **refix** an object, that is, atomically **unfix** and **fix** an object at a new place.

The **move...to** primitive was intended to be used for enhancing performance by colocating objects, and thus reducing the number of remote invocations between them. In contrast, **fix** was intended for applications where the location of an object was part of the application semantics. For this reason, we gave **move** weak semantics: a move was treated as a performance hint. The kernel was not obliged to perform the move if a problem was encountered with it; if, for example, the destination node were unavailable, **move** would do nothing silently. Moves were also queued, so that multiple move requests following one another would usually be batched, which in many cases gave us a huge (order of magnitude) performance improvement. Even if the move

succeeded, there was no guarantee that the object concerned might not immediately move somewhere else.

The **fix**, **unfix**, and **refix** primitives were designed for use when location was part of the application semantics, as when trying to achieve high availability by positioning multiple replicas on different machines, or when implementing a program that performed load sharing (an Emerald load sharing program was written later [68]).

We therefore gave **fix** much stronger semantics and implemented **fix** transactionally, so that after a successful **fix** the programmer could be sure that the object concerned was indeed at the specified location. An attempt to **move** a fixed object would fail, to avoid potential confusion as to whether **move** or **fix** had priority.

4.7.2 Moving Parameter Objects

As early as the *Getting to Oz* memo of April 1984 [69], we had decided that call by object reference was the only parameter passing semantics that made sense for mutable objects; this was the same semantics used by CLU and Smalltalk. However, we were also aware that call by reference could cause a storm of remote invocations, and for this reason invented two special parameter-passing modes that we called *call by move* and *call by visit*. The memo continues: “a new parameter passing mechanism we’ve considered is *call by move*, in which the invoked operation explicitly indicates that the parameter object should be moved to the location of the invoked object.” We saw *call by move* as giving us the efficiency of *call by value* with the semantic simplicity of consistently using *call by reference*. However, *call by move* was not always a benefit; although it co-located a parameter with the target object, it would cause any invocations from the call’s initiator to the parameter object to become remote, which could drastically reduce performance. The cost of the call would also be increased, albeit for smaller objects (less than 1 000 bytes) the cost was about 3.5% for *call by move* and 7% for *call by visit* [58, p. 131].

The Emerald compiler decided to move some objects on its own authority. For example, small immutable objects were always moved, because in this case the cost was negligible. In general, however, application-specific knowledge was required to decide if it was a good idea to move an object, and Emerald provided **move** and **visit** keywords that the programmer used to communicate to the compiler that a parameter should be moved along with the invocation. The use of these keywords affected locatics but not semantics: the parameter was passed by reference, as was any other parameter, but at the time of the call it was relocated to the destination node. The difference between **move** and **visit** was that, after the invocation had completed, a call-by-visit parameter was moved back to the source node when the invocation returned, whereas a call-by-move parameter was left at the destination.

Call by move was both a convenience and a performance optimization. Without *call by move*, it would still have been possible to request the move (using the **move...to** primitive), but that would have required the programmer to write more code and would not have allowed the packaging of parameter objects in the same network message as the invocation. There was also a *return by move* for result parameters.

4.7.3 Implementation

Whereas Eden objects had been implemented as whole address spaces, Emerald objects were data structures inside an address space, and so most of the implementation techniques that we needed had to be invented from scratch. A guiding principle was not to do anything that sped up mobility at the expense of slowing down local operations: the costs of mobility should rest on the applications that used it.

Moving the data structure representing an object was not conceptually difficult: we just copied it into a message and sent it to the destination machine. However, all of the references to objects in that representation were local pointers to object descriptors, and had to be translated to new pointers at the destination. To make this possible the kernel had to be able to distinguish object references from direct objects, which was achieved by having the compiler allocate all of the direct objects together, and putting a template in the code object that specifies how many direct objects and how many object references were in the object data area. A table that translated object references to *OIDs* was appended to the representation of the object, and the kernel at the destination used this translation table, in combination with its own object table, to overwrite the now-invalid object references with valid pointers to local object descriptors.

Emerald objects contained processes as well as data; this included their own processes and processes originating in other objects whose thread of control passed through the object. Whereas each object was on a single machine, Emerald processes could span machines. This could occur either because of a remote invocation, or because an object moved while a process was executing one of its operations. We realized that if we treated the execution stack of a process as a linked list of activation records, and each activation record as an object referenced by a location-independent object reference, then everything would “just work”: processes returning from the last operation on one machine would follow a remote object reference back to the previous machine.

Of course, in reality things were not quite so simple. Each activation record was not a separate object; instead, we allocated stack segments large enough to accommodate many activation records (the standard size was 600 bytes), and linked in a new segment only when an existing one filled up. This meant that when an object moved, we might have to split a stack segment into three pieces and then move the middle one. Also, stack segments were more complicated

that ordinary object data areas, for example, because they contained saved registers, and because the data stored in the activation record was constantly being changed by the running process.

An alternative to moving activation records with the objects to which they referred would have been to leave them in place until the executing process returned to them. We chose not to do this because it would have set up residual remote dependencies: if the machine hosting the activation record was unavailable, the executing process would be “stuck”. Although this might sound like an unlikely scenario, we thought that it would actually be quite common to move all the objects off of a machine so that it could be taken down for maintenance. If, in spite of moving all of the objects, the activation records were left behind, the computation would still be dependent on the machine that was down.

Finding out which activation records to move when an object migrates is a little bit tricky. Although each activation record has a context pointer linking it to an object, the object does not normally have a pointer back in the other direction. We considered adding one, linking a list of activation records from each object, but that would have almost doubled the cost of operation invocation. However, searching through every activation record whenever an object moved would have been prohibitively expensive. The compromise we adopted was to link objects to their activation records only when there was a context switch from one process to another. After all, it was only during a context switch that an object could move; in between two context switches many thousands of activation records would have been created and destroyed without any overhead.

We used a lazy technique to ascertain what data was on the current stack when a process was moved. This technique added no overhead to normal execution, placing it all on the move. The code object contained a *static template* that described the format of the current activation record for any given code address. As the name implies, static templates could be generated at compile time. We originally thought we would need a different template for each point in the code that changed the content of the stack, leading to a large number of templates. To avoid this we devised what we called *dynamic templates*, which described the *change* in the contents of the stack. However, Norm examined the problem carefully and found out that most temporaries pushed onto the stack could just as well be stored into temporary variables *preallocated* on the stack; because these variables could be reused throughout an operation, this caused no change to the stack layout and thus no change to the template. Moreover, there was no real storage cost: the stack needed to accommodate only the maximum number of temporaries simultaneously in use at any point in the operation.

Any other temporaries that were pushed onto the stack were full object references that included a pointer to the AbCon,

and so were self-describing. Thus, template information was not needed for these variables, and so dynamic templates were unnecessary. One static template, laid down by the compiler, was sufficient.

4.8 Failures

Emerald did not include a general-purpose exception handling mechanism. This was largely because Andrew was opposed to such mechanisms, and also because designing a good one would have distracted us from our goals. Because of Andrew’s thesis research [20], we were aware of the distinction between exceptions—special return values explicitly constructed by the program in known situations—and failures—which occurred when programs went wrong.

An Emerald **begin...end** block could be suffixed with a failure handler that specified what to do if one of the statements in the block failed, e.g., by asserting a false predicate, dividing by zero, or indexing a vector outside of its bounds. We did not regard a failure as a control structure for deliberate use, but as a bug that should eventually be fixed, and in the meantime survived. This meant that if there were some question whether or not an index was within the bounds of a vector, we expected the programmer to test the index before using it, rather than to have the indexing operation fail and then “handle” the failure. For this reason, Emerald did not include a mechanism for distinguishing between different kinds of failure.

If a failure occurred and there was no failure handler on any block that (lexically) contained the failing statement, the failure was propagated back along the call chain until a failure handler was found. Along the way, each object on the call chain was marked *failed*; this meant that any future attempt to invoke that object would fail. The language also provided a **returnandfail** statement so that an operation called with invalid parameters could cause its *caller* to fail without the invoked object itself failing.

4.9 Availability

As noted in Section 2, it was an explicit goal of Emerald to accommodate machine and network failures. At a given time, each mutable Emerald object was located on exactly one machine. Thus, if a machine crashed, the objects on it would become unavailable, and it would be temporarily impossible for another object to invoke them. We saw unavailability as a common and expected event, and felt that distributed programs had to deal with it, so we provided a special language mechanism to handle unavailability.

Our view was that unavailability was quite different from failure. The availability of an object was not like the property of an index being within bounds: programmers could not test for availability before making each invocation, because availability was a transient property of the environment rather than a stable property of the program state.

Emerald allowed programmers to attach an *unavailable handler* to any block. This handler specified what to do when, due to machine crashes or communication failures, an invocation could not be completed or an object could not be found. For example, if an object tried to invoke a name server and that name server was unavailable, it could try a second name server:

```
begin
  homeDir ← nameServer1.lookup[homeDirectoryName]
end
when unavailable
  homeDir ← nameServer2.lookup[homeDirectoryName]
end unavailable
```

An *unhandled unavailable* event was treated as a failure. So, if *nameServer2* were unavailable, the invocation of its *lookup* operation would fail and the object containing the above code would also fail.

4.10 Kernel Structure and Implementation

As mentioned in Section 1.5, our use of the term *kernel* followed the tradition of Concurrent Pascal and Eden. In all these systems the term meant the run-time support software that was responsible for loading programs, managing storage, performing I/O, creating and administering processes, and performing remote invocations. The Concurrent Pascal kernel actually ran on a bare machine. The Eden kernel was a UNIX process, and implemented each Eden object as another UNIX process; this led to excessive storage consumption (minimum size of an object was 300 kBytes) and execution time (invocations between Eden objects on the *same* machine took on the order of 100 ms). We wanted substantially less overhead both in storage and execution time, so we decided to implement the kernel in a single address space as a single UNIX process within which all activity remained. This saved us from expensive UNIX process boundary crossings and allowed us to make object invocations almost as fast as procedure calls in C. We handled our own storage allocation, so we were in complete control of storage layout; this let us implement mobility and prepared the system for garbage collection.

The kernel was written in C, which we considered to be an advanced assembler language: it allows detailed and efficient access to data structures including the execution stacks and let us build an invocation mechanism optimized for performance. As mentioned in Section 3, we wrote stylized C code designed to generate assembly language programs that were as efficient as hand-coded assembler, but with the advantage that portability was substantially better. Portability was a concern to us. One of the reasons that Eden had not seen any use outside the University of Washington was that it was not portable: Eden required a modified version of the 4.1 BSD UNIX kernel and included much assembler code. We wanted Emerald to see wider use, so we strove to minimize dependence on assembly code and programmed using as generic a subset of the UNIX API as possible.

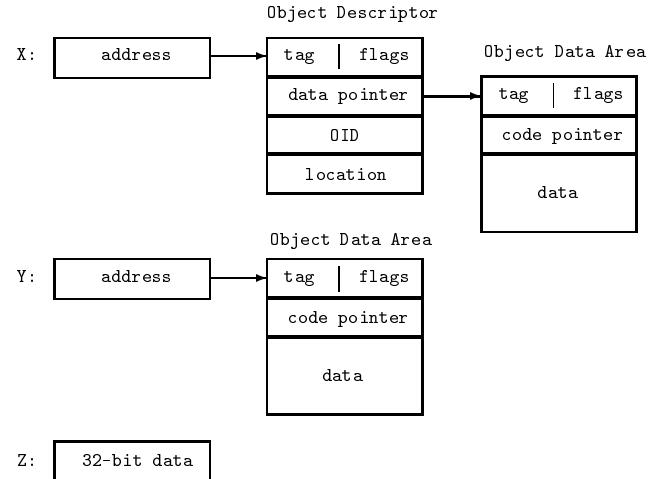


Figure 9: Emerald object layouts from Eric's thesis [58]. X is the two-level structure used for *global* objects, Y is the one-level structure used for *local* objects, and Z shows a *direct* object.

dence on assembly code and programmed using as generic a subset of the UNIX API as possible.

4.10.1 Object Layout and Object IDs

In the language, each object could potentially be accessed from a remote machine. So, in principle, every object required a globally unique object identifier, known as an *OID*. Because Emerald was developed on a local area network, we initially implemented *OIDs* as 8-bit machine numbers (we used the bottom 8 bits of the machine's IP address) concatenated with 24-bit sequence numbers. In a subsequent wide-area network version of Emerald, the *OID* was expanded to be the full-32 bit IP address and a 32-bit sequence number [80]. To avoid wasting storage and *OID* space on objects that would never be referenced remotely, we did not actually allocate an *OID* until a reference to the object was exported across the network.

We expanded the use of *OIDs* to other kernel data structures, which made those structures remotely addressable using the same mechanism as for objects. For example, the structure containing the code for an object was given an *OID* by the compiler (see Section 4.10.4). For this reason, in this section on kernel implementation we use the term *object* to mean the representation of either a real Emerald object or an object-like kernel data structure.

Figure 9 shows the layout of kernel data. There were two representations for non-primitive objects: a two-level scheme (X) for global objects and a one-level scheme (Y) for local objects. The first word of every kernel data area had a standard format: a tag that identified the data area and a number of tag bits, including the frozen bit and an indication of the representation scheme. The two-level storage scheme represented an object by the local address of an *Object De-*

scriptor. The Object Descriptor indicated whether or not the object was locally resident. If it was, the local address of the *Object Data Area* was stored in the Object Descriptor. If the object was not resident, the Object Descriptor contained the *OID* for the object and a hint as to the current location of the object. In the one-level representation, the Object Descriptor and Object Data Area were merged together; we did this to promote efficiency in access, allocation and storage space, but only for objects that could never be accessed remotely.

Each *OID* was entered into a hashed *object table* that mapped *OIDs* to Object Descriptors. This meant that an *OID* arriving over the network from another machine could be mapped to the corresponding Object Descriptor, if one already existed. This made it easy to ensure that a given object had no more than one Object Descriptor on each machine. For mutable objects, only one of the Object Descriptors in the whole Emerald system would normally reference an Object Data area. Moving an object from one machine to another therefore meant copying the Data Area from one machine to the other, and then changing a single address in the Object Descriptor on the source and destination machines. While the object was being moved, it would have a data area on both machines, so the object was flagged as *frozen* to prevent the object's operations from modifying the data. Immutable objects could be replicated; they were represented by an Object Descriptor *and* an Object Data Area on each machine where there was a reference to the object.

4.10.2 Kernel Concurrency: Task Management

The Emerald kernel needed to keep track of multiple concurrent activities, for example, an ongoing search for an object, a remote invocation, and the loading of some code over the network. At the time, good thread-management libraries were not available, and instead of writing our own thread package we decided to use event-driven programming, an idea inspired by the MiK kernel [93]. We built a single-threaded kernel that serviced a ready queue of kernel tasks, which we made sure were atomic. These tasks were generated from many sources, for example, the arrival of a invocation request message, a remote code load, or a *bootup* message from another Emerald kernel. Many events arrived in the form of a UNIX signal. For these, the signal handlers merely set a *signal-has-occurred* bit; they were not allowed to touch any other kernel data structures. To make sure that signals were not dropped, kernel tasks were required to be short and not to perform any blocking operation. If a task needed to wait, it had to do so by generating a continuation task that would run when the appropriate event arrived, and then terminating itself. Synchronization around the *signal-has-occurred* bit was complicated and took a lot of low-level hacking; the details can be found in a technical report [53].

Each waiting kernel task was potentially linked into a dependency graph: if a task *B* was waiting for another task *A* to complete, *B* would be linked from task *A*. When *A*

completed, it would put all tasks waiting for it onto the ready queue, unless they were also waiting for another uncompleted task. For example, the arrival of a moving object would generate an *object install* task, but if the code for that object were not already resident, a *code load* task would be generated to find and load the code. The *object install* would be dependent on the *code load*. When the *code load* completed, the *object install*'s dependency on the *code load* task would be removed and, if the install task had no more dependencies, it would be put on the ready queue.

4.10.3 Choice of Networking Protocols

Eden was built on a message module that used datagrams and did not provide reliability. For Emerald, in contrast, Eric modified the Eden Message Module to provide reliability by using a sliding window protocol taken from Tanenbaum's first *Computer Networks* book [101]. (In the process he found an error in the protocol.) The importance of a truly fault-tolerant communication protocol had become apparent to Eric during a demo of Eden, when an object on one machine had reported that another object was inaccessible. This was despite the fact that it was obviously alive because it was displaying output on another screen, and was observed as alive by an object on a third machine. After this demo Eric spent a lot of time researching fault-tolerance and recovery and worked hard to ensure that the low-level protocols would be quite robust.

We considered using TCP instead of UDP, but the version of UNIX we were using allowed only a very limited number of open connections, both because UNIX had a low limit on the number of open file descriptors and because TCP connections used a significant amount of buffer space. This meant that full machine connectivity would require us to open and close TCP connections frequently, leading to excessive overhead and high latency for remote invocation. We also wanted to have full information about when nodes failed to respond to messages, so that we could declare them dead according to our own policy. In Eden, we had noticed that nodes could be considered *up* even if they had died some time previously; even worse, nodes could be considered *dead* even though they were *up*. We realized that no failure detector could be completely reliable, but we felt that because we knew more about the Emerald system than did the generic implementation of TCP in the UNIX kernel, we could do a better job.

Eric's implementation of reliability on top of UDP strove to reduce the number of network messages to the absolute minimum. We used just two Ethernet messages for a remote invocation: the acknowledgment message, which was necessary for reliability, was piggybacked on the next interaction. Thus, a series 100 remote invocations to the same destination would require 201 Ethernet packets: 100 invoke-request packets, 100 invoke-return packets, and, after about 1 second, a single acknowledgment packet—the remaining 199 ACKs would be piggybacked onto the other packets.

4.10.4 OIDs for Compiled Entities

The compiler generated a file for each object constructor and each type. For the purpose of loading and linking, the compiler assigned a unique *OID* to each file, and any reference from one file to another used the *OID*. The compiler had its own pseudo-machine number, so in effect it had control over its own *OID* space, which was disjoint from the space of “real” objects.

Because types were objects, they needed to have *OIDs*; because types were immutable, it was convenient to use the same *OID* across multiple compilations of the same type. (Recall that Emerald types contain only operation signatures, and not executable code.) Two types were the same even if the Emerald sources that defined them were different in insignificant ways; for example, if the order of operations was different in two versions of a type, or if they used different names for themselves or their component types. We also gave *OID* to Strings used as operation names; Strings were also immutable objects. The purpose of this was to speed-up the comparison of operation names when deciding if one type conformed to another.

The initial implementation of type *OIDs* and String identifiers was a file on the local network file system, which the compiler treated as a global database, and in which it stored a canonical form of the type definition. Of course, we were well-aware that this implementation would not scale to a global Emerald network, but at the time we had only five machines, and it worked quite well. Our longer-term plans involved using fingerprints of the type definitions, a technique that we had recently heard about from colleagues at DEC SRC, where it had been used in the Modula-2+ system. Fingerprinting was originally invented by Michael Rabin [88]; the techniques that made it practical were developed by Andrei Broder at SRC in the mid 1980s, although they were not published until much later [31]. Broder’s Fingerprinting algorithm offered strong probabilistic guarantees that distinct strings would not have the same fingerprint. We never actually got around to using fingerprints; when Norm used Emerald in distributed systems classes, he instead implemented *Type* to *OID* and *String* to uid servers as Emerald applications.

4.10.5 Process Implementation

Processes were implemented quite conventionally using one stack for each process and one activation record for each invocation. However, our stacks were segmented: a stack could consist of multiple stack segments, each of which contained some number of activation records. When returning off the “bottom” of a stack segment, the run-time system would check to see if the process had more stack segments, in which case the process would continue execution using the top activation record of the next segment.

There were two reasons for segmenting a stack. First, segmentation allowed us to allocate a small stack and then expand it dynamically, thus saving space and letting us support a substantial number of concurrent processes. Second, segmentation allowed the process to make remote invocations. The links between stack segments were object references, and each link could thus point to a segment on another machine. When a process performed a remote invocation, it simply continued execution on the remote machine using a new stack segment containing the activation record for the invoked operation. When the operation returned off the bottom of the stack segment, the invocation results were packaged up and sent back to the originating stack segment.

4.10.6 Dynamic Code Loading

Because Emerald objects shared a single address space, introducing a new kind of object meant loading new code into the address space. The loader had to perform “linking”: references to abstract or concrete types in the loaded code had to be replaced by references to the Object Descriptor of the appropriate object.

We adopted the principle that *all* of the code files needed for an object to execute had to be loaded before execution was allowed within the object. This was an example of a general philosophy of minimizing remote dependencies: as much as possible we wanted to insulate objects from failures of remote machines. By aggressively “hoarding” all the code files required by an object we ensured that the object, once it started execution, would never stall waiting for code to load, or fail because it needed code located on a machine that had become unavailable. Loading all of the code before execution started also meant that there was no need for dynamic *code-loaded* checks that would have introduced overhead into all programs, even those that didn’t use distribution. This would be in violation of our *no-use, no-cost* principle mentioned in Section 2.

5. Applications and Influences of Emerald

Immediately after the implementation of Emerald was completed, indeed even before it was completed, the Emerald team dispersed. In January 1987, Norm graduated and took up a faculty position at the University of Arizona; Eric went back to Denmark in February 1987 and taught at the University of Copenhagen while finishing his thesis. Andrew joined the distributed systems group of Digital Equipment Corporation in December 1986. Only Hank grew roots at Washington, where he is now department chair. Nevertheless, in addition to the initial batch of research papers, Emerald has lived on in several forms. It has been used in teaching and graduate student education and as a basis for subsequent research. Emerald also influenced the design of successor systems. In this section we summarize what we have been able to discover about the use and influences of Emerald after the initial implementation effort.

5.1 Applications of Emerald

Emerald has been used in teaching and research at various universities; in most cases Emerald was carried to other sites by those who had become familiar with the language at Washington.

Norm has used Emerald at the University of Arizona and at the University of British Columbia (UBC) in teaching graduate classes on distributed operating systems. Three M.Sc. students at UBC did their theses on aspects of Emerald including generational garbage collection [48], porting the language to the embedded processor on a network interface card [86], and improving reliability by grouping objects [45].

Eric has used Emerald at the University of Copenhagen for teaching at both the graduate and undergraduate levels. From 1994–1997, Emerald was the first object-based language taught to incoming students (their first language being ML). During these four years approximately 1000 students used Emerald for about three months, including a three-week-long graded assignment where they were to develop and write a large (1000–2000 lines of Emerald) program, such as a ray tracer. They did not use any of the distribution or concurrency facilities, but focused on the pure object-based part of the language. During the 1990s Emerald was also used to teach distribution in an introductory graduate distributed systems course at DIKU.¹¹ Students were required to write small (200–300 lines of Emerald) programs to implement a distributed algorithm such as election or time synchronization. About fourteen Master's students have to varying degrees based their Master's theses on Emerald [5, 54, 55, 56, 63, 65, 68, 78, 80]. In addition, two Ph.D. students conducted research based on Emerald: Niels Juul worked on distributed garbage collection [60] and Niels Larsen worked on transactions [66].

Andrew used Emerald in research while at Digital Equipment Corporation. It was used to build one of the first distributed object-oriented applications to run on Digital's internal engineering network, a distributed object-oriented mail system implemented by a summer student in 1987. This student (Rajendra Raj) later went on to develop a programming environment for Emerald [89], called Jade, that was the subject of his doctoral dissertation [91].

Other University of Washington students took Emerald with them to other parts of the globe. Ewan Tempero used it in research at the Victoria University of Wellington. Results include two theses: Neal Glew's B. Sc. Honours thesis [47] and Simon Pohlen's M. Sc. Thesis [87].

Emerald was also used in research at the University of Mining and Metallurgy (UMM) in Kracow, Poland. The research focused on multicast group communication, and was ini-

tiated by Przemek Pardyak and Eric Jul during a visit by Przemek to Copenhagen in the early 1990s. Przemek subsequently worked with Krzysztof Zielinski at UMM, resulting in two papers [82, 83] and a Master's thesis [81]. The thesis was awarded third prize in the annual Polish competition for best engineering thesis. Along the way, they also published an overview of Emerald in Polish in a book on distributed systems; this article also appeared in the main Polish CS magazine *Informatyka* [84].

5.2 Influences of Emerald

Emerald has influenced succeeding languages designs in two main areas: support for distributed objects and advanced type systems. Because the goal of Emerald was to innovate with distributed objects, the language's influence on distribution is unsurprising. (What is perhaps surprising is that in the more than twenty years since we implemented Emerald, no other language has adopted mobile objects with similar thoroughness.) In contrast, Emerald's influence on type systems was neither intended nor expected. At the time we saw our innovation here as minor: we were going to do only what was required to keep the language as small as possible. The idea of conformity has been quite widely adopted. In an interesting example of feedback, both abstract types (here called protocols) and conformity feature heavily in the ANSI Smalltalk standard. The 1997 final draft says

The standard uses a particular technique for specifying the external behavior of objects. This technique is based on a protocol, or type system. The protocol-based system is a lattice (i.e. multiply inherited) based system of collections of fully-specified methods that follow strict conformance rules [6, p. 6].

However, Java adopted nominal rather than structural typing (see Section 4.2.3), and only partially separated types and implementation: a Java interface is a type, but a Java class is both an implementation *and* a type.

Several operating systems have followed Eden and Emerald in providing mobile objects; notable among these is SOS [94], which in addition allows an object to be fragmented across several machines. However, SOS provides only minimal language support, so its objects and proxies must be manipulated by explicit operating system primitives. (SOS does use a compiler extension to simplify the process of obtaining a proxy for a remote object.) One of the early derivatives of Emerald's object model and type system was the Advanced Networked Systems Architecture (ANSA) distributed computational model [106]. In its turn, ANSA was one of the influences behind the Open Software Foundation's Distributed Computing Environment initiative, and perhaps more significantly, a contributor to the ISO Basic Reference Model of Open Distributed Processing ISO 10746 [46]. The architecture described in part 3 of this standard includes a type system very strongly based on

¹¹ DIKU is the department of Computer Science at the University of Copenhagen.

Emerald, extended with support for multiple terminations, streams and signals—ideas taken from ANSA. In particular, Emerald’s type conformity rules are alive and well in Section 7.2.4.3, and Annex A gives a set of subtyping judgments including top and bottom types strikingly similar to those used to formalize the Emerald type system [26].

Andrew Herbert, Chief Architect of the ANSA efforts, describes how the ANSA architecture was influenced by Emerald.

Alongside the ANSA architecture we maintained a number of software prototypes. The first version of ANSAware was called “Little DPL” and was pre-processor based... A separate strand of development was “Big DPL”, which borrowed very heavily from Emerald. This was the foundation for the input to ISO and also to OMG CORBA. It took the view that you could do a complete language based on “distributed objects”, where an object could have a number of “interfaces” (which were named typed instances and could be static or dynamic). Our invocation model was that an interface defined a bunch of methods and that each method had a bunch of “terminations”, one of which generally was regarded as the “normal return” and the others as “exceptions”. All parameter semantics were call by reference. Immutable objects could of course be copied, and we had an object migration facility. Migratability and other “ilities” were object properties and objects would generally have a control interface through which the “ility” could be exercised.

Big DPL was never well-enough developed and not mainstream enough to compete with the C++ network object systems that came along around the same time... Over time it shifted from a separate “toy language” to DIMMA, yet another C++ distributed object system, but which carried forward explicitly the DPL type system. The final evolution was the “FlexiNet” system written in Java, which had a hybrid Emerald/Java object model and Java type system...

In summary Emerald had a lot of impact on us, and through us on other bodies.¹²

Andrew Watson started working at ANSA in December 1989. At that time both the ANSA computational model and the language that realized it (DPL) were well established. There were several technical differences between Emerald and DPL, principally that DPL allowed multiple interfaces per object, that each operation in DPL could have multiple named outcomes (with different types), and that DPL had absolutely no primitive types built in. Watson writes:

However, leaving these differences aside, the influence of Emerald on DPL was obvious—especially

¹² Andrew Herbert, personal communication.

the conformance-based type checking and type inferencing in the DPL language. It was this type system that I chose to work on, and in particular the problem of typing constructors for collections of homogeneous and heterogeneous interfaces, and how to avoid having every object carry around its type information at run time via an Emerald-style “getInterface” operation. I did come up with a simple modification to DPL that added a separate, opaque run-time type representation which would only be created when required by the programmer—this would dramatically reduce the need for the run-time system to create and transmit run-time type tokens... [but the implementation of this modification] was never finished.¹³

Emerald also had an impact on the design of the Guide system and language at INRIA in Grenoble, France. Sacha Krakowiak writes:

One important source of inspiration for the design of the Guide language has been Emerald, a distributed object-oriented language developed as a follow-on project to Eden. The two main features of the design of Emerald that directly influenced Guide were the separation between types and implementations, and the definition of type conformity (through covariant and contravariant relations). This definition had been proposed by Cardelli in 1984 [35], but we did not know that work.¹⁴

Emerald’s closest descendant in the family of distributed object systems is probably the Network Objects system described and implemented by Birrell, Nelson, Owicki and Wobber [12, 13]. The authors write: “We have built closely on the ideas of Emerald [59] and SOS [94], and perhaps our main contribution has been to select and simplify the most essential features of these systems. An important simplification is that our network objects are not mobile” [12, Section 2]. Instead, the Network Objects system provided what the authors called “powerful marshaling”: the ability to send a copy of an arbitrarily complex data structure across the network. They write [13, p. 10]:

We believe it is better to provide powerful marshaling than object mobility. The two facilities are similar, because both of them allow the programmer the option of communicating objects by reference or by copying. Either facility can be used to distribute data and computation as needed by applications. Object mobility offers slightly more flexibility, because the same object can be either sent by reference or moved; while with our system, network objects are always sent by reference and other objects are always sent by copying. However, this extra flexibility doesn’t seem to

¹³ Andrew Watson, personal communication.

¹⁴ Sacha Krakowiak, personal communication.

us to be worth the substantial increase in complexity of mobile objects. For example, a system like *Hermes* [25], though designed for mobile objects, could be implemented straightforwardly with our mechanisms.

(As might be expected, we do not entirely agree with these conclusions; see the discussion in Section 6.)

Emerald was also influential in the development of the distributed systems support provided by Java and Jini, although here the influence was more indirect. Indeed, at first blush the Java RMI mechanism seems to be the antithesis of Emerald’s remote object invocation, because RMI distinguishes between remote and local invocations. This is not accidental: Jim Waldo and colleagues at Sun Laboratories authored a widely referenced technical report that argues that the “unified object” view espoused by Emerald is fundamentally flawed [104]. The account here is largely based on material supplied by Doug Lea.

Java RMI was most directly influenced by CORBA and Modula-3’s Network Objects. In the summer of 1994, James Gosling’s *Oak* language for programming smart appliances was retargeted to the World-Wide Web and soon thereafter renamed Java. Jim Waldo’s group were asked to look into adding an object-oriented distributed RPC of some form. At the same time, some people who had been in the Sun *Spring* group (who were also contributors to CORBA) were looking into the alternative approach of just providing Java with a CORBA binding. Both of these approaches had their advocates. Some people were excited by the “Emerald-ish” things one could do with extensions of Waldo’s approach. Also, Cardelli’s paper on *Obliq* had just appeared [37]. Obliq objects are implemented as Modula-3 network objects, and so are remotely accessible. While Obliq *objects* are fixed at the site at which they are created, distributed lexical scope allows *computations* to roam over the network. Although the people at Sun regarded Obliq as a thought experiment, it clearly demonstrated the limitations of CORBA, which was unable to express the idea of a computation moving around the network; thus Obliq provided fuel for the argument that Java needed a more powerful remote communication system. Doug Lea writes: “I wanted a system that was not only usable for classic RPC, but also for extensions of the things that I knew to be possible in Emerald. My bottom line was that I insisted it be possible to send a Runnable object to a remote host so that it could be run in a thread.” After much deliberation, a committee at Sun chose Waldo’s approach, and many of the researchers in Waldo’s group transitioned from Sun Labs to the Java production group, and built what is now known as Java RMI. After RMI was released, they moved on to develop further what they had earlier been working on in Sun Labs, which turned into Jini [7].

As it exists today, RMI supports neither Emerald-style mobile objects nor Obliq-style mobile processes, and remote

invocations are not location transparent, because different parameter-passing mechanisms can be used for local and remote invocations of the same object. But many of these restrictions are the result not of limited vision, but of compromises that had to be made to fit RMI over the Java language, whose specification was at that time largely fixed. Emerald was a force for a more adventurous design, and the ultimate decision to go with RMI rather than a CORBA binding was influenced by reading or hearing about Emerald’s capabilities—these were continually brought up as the kinds of things that a forward-looking language ought to support. Jim Waldo writes:

The RMI system (and later the Jini system) took many of the ideas pioneered in Emerald having to do with moving objects around the network. We introduced these ideas to allow us to deal with the problems found in systems like CORBA with type truncation (and which were dealt with in Network Objects by doing the closest-match); the result was that passing an object to a remote site resulted in passing [a copy of] exactly that object, including when necessary a copy of the code (made possible by Java bytecodes and the security mechanisms). This was exploited to some extent in the RMI world, and far more fully in the Jini world, making both of those systems more Emerald-like than we realized at the time.¹⁵

After the turn of the century, increasing use of mobile devices lead to an interest in languages that supported loosely connected devices. The ideas behind Emerald inspired Walsh’s thesis work on the Taxy Mobility system [105]. Walsh introduces Emerald-like objects into Java and discusses the idea of combining weak and strong mobility. Similarly, the work of De Meuter takes a critical look at strong mobility as present in Emerald and proposes alternative, weaker, mobility mechanisms [42].

5.3 Later Developments

Various research projects have built on Emerald: to complete the implementation, to enhance the language, and to take Emerald in new directions.

Garbage collection. We realized early on (see the minutes from 18 March 1985) that Emerald would require a garbage collector, but we also realized that the implementation of the collector could be deferred until after Norm and Eric had completed their theses. In practise, Emerald did quite well without a collector. A prime reason was that the implementation was stack based and so only rarely would the implementation generate a temporary object and not be able to deallocate it.

Eric’s Ph.D. dissertation [58] contains a chapter on the design of a distributed, on-the-fly, robust, and comprehensive

¹⁵ Jim Waldo, personal communication.

garbage collector for Emerald, but there was no attempt to implement it at that time. However, Eric did prepare the Emerald prototype for the implementation of a distributed collector at some time in the future, concluding that the implementation effort would be worth a separate Ph.D. This is actually what happened: Niels Christian Juul implemented the collector for his doctorate [61, 60]. A strong point of this on-the-fly collector was that it was possible to start the collector and have it run to completion without requiring that all machines be up at the same time. In the contemporary demonstration of the collector, no more than 75% of the machines were up at any given time—we felt that this was a very robust collector.

In 2001-2002, two Master’s students developed a non-comprehensive collector that could collect smaller areas of the distributed object graph and thus could reclaim garbage while parts of the system was down [55].

Emerald as a general-purpose programming language. Those of us involved in the development of Emerald naturally saw it primarily as a language for distributed programming. It took an “outsider” (Ewan Tempero) to make us realize that Emerald was an interesting language in its own right, even without its distribution features. This realization led to a paper that described the language in this light, emphasizing Emerald’s novel object constructors and types [90].

The Hermes project at Digital Equipment Corporation. Andrew started the Hermes project shortly after he moved to Digital in December 1986. (This project is unrelated to Rob Strom’s Nil project at IBM, which also used the name Hermes in its later years [99].) The idea was to implement as many of Emerald’s distributed object ideas as possible in a conventional programming language (Modula-2+), but also to find a way to make the implementation scale to systems of around fifty thousand nodes, the size of Digital’s internal DECnet-based Engineering Network at that time. The project was a technical success [24, 25] but did not have much influence on future products because, in spite of the company’s early lead in networking, Digital was never able to make the transition to shipping distributed systems products.

Types. Norm and Andrew continued to work on types during the period from 1986 to 1989, after both of them had left Washington. Much of this work was unpublished, although not for want of trying. The target of our publication efforts was PoPL; at that time the PoPL community, based as it was in traditional single-machine programming language design, was developing ideas about the purpose of types and how they should be discussed that were much more conservative than those we had arrived at though our work in distributed systems. (To this day there is an enormous resistance from some members of this community to even admitting that something that might need to be checked at run time

could even be called a “type”.) The essence of our work was captured in a widely-cited January 1991 joint Arizona/CRL technical report [26], which had a major influence on the ANSA architecture (see Section 5.2).

Gaggles. One of the deficiencies of Emerald’s pure object-based approach is that not everything is most conveniently represented as an object. In particular, in a distributed system striving for high availability, resources must be replicated: the collection of replicas is not itself an object. Of course, it is possible to put the replicas *inside* an object, such as a Set, but this recreates a single point of failure. Gaggles resolve this problem: a Gaggle is a monotonically-increasing set of replicas that can be treated as a single object with “anycast” semantics [27]. An undergraduate student from Harvard, Mark Immel, undertook the implementation.

Multicast invocations. As mentioned in Section 5.1, Przemek Pardyak [82, 83] extended Emerald with facilities for multicast invocation. In his system, one could make an invocation of a group of objects, and either take the first answer, or require all answers.

Wide-area Emerald. In 1992-3, two Master’s students at DIKU implemented a Wide Area Emerald [80]. The implementation was modified to support machines on the internet in general rather than on a LAN. They increased the size of OIDs and had to spend a lot of time tuning the transport-layer protocols: their initial move of an object around the globe via seven machines took about 15–20 minutes, because the transport layer was optimized for LAN service. Eventually, moving an object around the world took only a few seconds.

Ports to various architectures. Between 1987 and 1994, Emerald was ported from the original VAX architecture to SUN 3 (Motorola 68000), SUN 4 (SPARC) [75], and Digital’s Alpha [66]. Tired of retargeting the compiler, Norm also developed a byte-code compiler and a virtual machine, thus allowing objects to move from one platform to another.

Eclipse plugin. In 2004, IBM funded a small project at the University of Copenhagen to implement an Eclipse plugin for Emerald. A student did the implementation during 2004–2006; the source code for the plugin—and for Emerald in general—is available at <http://sourceforge.net/projects/emeraldlanguage>.

Heterogeneous Emerald. Emerald’s clear separation of language concepts from implementation means that the semantics of Emerald objects have a rigorous (if informal) high-level description. Any implementation of an Emerald object thus must define a translation of the language’s high-level semantic concepts into a low level representation. This means that given two different implementations of an Emerald object for two different low level architectures, it should, in principle, be easy to remap the representation of an Emerald object on one architecture to the representation of that same object on another architecture.

Would this principle work out in practice? To answer that question, between 1990 and 1991, two graduate students (Povl Koch and Bjarne Steensgaard Madsen) worked with Eric to develop a version of Emerald that ran on a set of heterogeneous machines: VAX, SUN 3, and SUN SPARC. This mainly involved remapping inter-machine communication. However, object mobility also posed significant problems because an object could contain an active Emerald process. We introduced the concept of a *bus stop*, a place in the Emerald program where mobility could occur and where every implementation had to provide a consistent view. Bus stops are frequent; they are present after essentially every source-language statement. We then added so-called *bridging code* at each bus stop that could translate the state of an Emerald process or object from one architecture to another. The implementation was simplified because Emerald already had the concept of stopping points where mobility could take place.

To our knowledge, no one has since built a heterogenous object system allowing not only objects but also executing threads to move between architectures at almost any point in the program with *no* decrease in performance after arrival on the new architecture. That is, an Emerald process that moves, e.g., from a VAX to a SUN 4 SPARC and back will, when running on the SPARC, execute at the same speed as if it had originated there, and it will also execute at the original VAX speed after returning to a VAX. Cross-architecture performance measurements showed that remote invocations and object (and process) mobility took about twice as long as in the homogeneous case, mainly due to the large amount of work caused by checking for big-endian to little-endian translations, and byte-swapping if necessary. Note, however, that the programs locally always ran at full speed unimpeded by the facilities for heterogeneity.

Eric presented the heterogeneous Emerald work at a work-in-progress session at SOSP in 1993, and generated significant attention. This caused Eric to contact one of the students, who in the meantime had joined Microsoft Research. The result was a paper presented at SOSP in 1995 [97]. At the time, Marc Weiser was advocating the then-controversial idea that researchers should make their code publicly available, so that anyone so motivated could verify the claimed results. As a consequence, the Heteogeneous Emerald implementation can be found on the 1995 SOSP CD.

Databases, Queries, and Transactions A bright graduate student, Niels Elgaard Larsen, integrated databases into Emerald as his Master's thesis project [65]. He later integrated transactions into Emerald as part of his Ph.D. project [66].

6. Retrospective

The Emerald project never came to a formal conclusion; it simply faded away as the members of the original team

became interested in other research. Nevertheless, time has given us a certain perspective on the project; here we reflect on what we did and what we might have done differently.

We are all proud of Emerald, and feel that it is one of the most significant pieces of research we have ever undertaken. People who have never heard of Emerald are surprised that a language that is so old, and was implemented by so small a team, does so much that is “modern”. If asked to describe Emerald briefly, we sometimes say that it’s like Java, except that it has always had generics, and that its objects are mobile.

In hindsight, if we had had more experience programming in Smalltalk, Emerald might have ended up more like Smalltalk. For example, we might have had a better appreciation of the power of closures, and have given Emerald a closure syntax. Instead, we reasoned that anything one could do with a closure one could also do with an object, not really appreciating that the convenience of a closure syntax is essential if one wants to encourage programmers to build their own control structures.

6.1 The Problem with Location Independence

Emerald’s hallmark is that it makes distribution transparent by providing location-independent invocation. However, there is definitely a downside to this transparency: it becomes easy to forget about the importance of placing one’s objects correctly. Shortly after Eric started using Emerald in his graduate course on distributed systems, a student showed up at his office with an Emerald program that appeared to hang; the student could not find any bug in the program. Eric suggested turning on debugging and started with traces of external activities. Streams of trace output immediately showed up: the program was not hung but was executing thousands of remote invocations. What had happened was that the student had omitted an **attached** annotation on a variable declaration, which meant that he was remotely accessing an array that should have been local. Thus the program was running three to four orders of magnitude slower than anticipated. The good news was that Emerald’s location independent invocation semantics means that the program was still correct; the bad news was that it ran more than a thousand times too slowly. After adding the omitted **attached** keyword the program ran perfectly!

6.2 Mobile Objects

Since we started working on Emerald, objects have become the dominant technology for programming and for building distributed systems. However, mobile objects have not enjoyed a similar success. Why is this so? The argument against mobile objects goes something like this.

The simplicity of object orientation arises because objects are good for modeling the real world. In particular, objects enable the sharing of information and resources inside the

computer, just as in the real world. Understanding object identity is an important part of these sharing relationships.

Mobile objects promise to make that same simplicity available in a distributed setting: the same semantics, the same parameter mechanisms, and so on. But this promise must be illusory. In a distributed setting the programmer *must* deal with issues of availability and reliability. So programmers have to replicate their objects, manage the replicas, and worry about “one copy semantics”. Things are not so simple any more, because the notion of object identity supported by the programming language is no longer the same as the application’s notion of identity. We can make things simple only by giving up on reliability, fault tolerance, and availability—but these are the reasons that we build distributed systems.

Once we have to manage replicas manually, mobile objects don’t buy us very much over Birrell’s “powerful marshaling” [13]: making a copy of an object at a new location. They do buy us something, but the perception is that it is not worth the implementation cost.

We feel that this argument misses the point, and for two reasons. First, the implementation cost is not high. Eric and Norm may be smart, but they are not *that* smart; if they could figure out how to make objects mobile and efficient in the mid-1980s, it should not be hard to reproduce their work in the twenty-first century. Indeed, we believe that implementing mobility right once is simpler than implementing the various *ad hoc* mechanisms for pickling, process migration, remote code loading, and so on.

Second, we now know enough about replication for availability to design a robust mechanism like Gaggles [27] to support replicated objects in the language. In the presence of such a mechanism, object identity once again becomes a simple concept: in essence, the complexity of the replication algorithm has been moved *inside* the abstraction boundary of an object. This does not make it any simpler to implement, but does permit the client of the complex replicated abstraction to treat it like any other object.

There has also been a long-running debate [62, 74] that questions whether object identity should be a language-defined operation at all, because it breaches encapsulation. Conceptually, we have a lot of sympathy for this position, but pragmatically we know that it is important to support the operations of identity comparison and hash on all objects if we want to do efficient caching—and caching is a very important technique for building efficient distributed systems. A compromise would seem to be in order. Andrew advocated just this in 1993: equality and hashing of object identity should be fast, primitive operations that do not require fetching an object’s state, but the programmer should be able to allow or disallow these operations on an object-by-object (or perhaps class-by-class) basis [23].

6.3 Static Typing

With the benefit of hindsight, static typing may have been a mistake: static types bought us very little in the way of efficiency, and cost us a great deal of time and effort in developing the theory of bounded parameterized types. The one place where static types do buy efficiency is with primitive types such as integers, but this is not because of the types themselves. The efficiency gain arises because we break our own rule and confound implementation (of an integer as a machine word) with interface. In other words, integers are efficient only because we forbid the programmer to write an alternative implementation of the *integer* type.

In the normal case of an object of user-defined type that is the target of an invocation, we know that the dynamic type of the object conforms to the static type of the identifier to which it is bound. This guarantees that the invocation message will be understood at run time, but it does *not* help us to find the right code to execute. Finding the code must still be done dynamically, because the implementation of the object is generally not known until run time. Of course, in many cases the implementation will be fixed at compile time, in which case method lookup can be eliminated altogether. But the presence of type information did not help us to implement this optimization: it relies on dataflow analysis.

Emerald’s dynamic type checks rely on the compiler telling the truth: the type of an object is encoded in it as a **type** when the object is constructed, based on the information in its object constructor. If the compiler lied, type checking might succeed, but operation invocation still fails. We considered certifying compilers and having them sign their objects, but Because no one ever wrote a hostile Emerald compiler, signatures were never implemented. This is one place where Java does something simpler and more effective than Emerald: Java byte codes are typed, and code arriving from another compiler can be type-checked when it is loaded. This is an idea that we might well have used in Emerald if we had been aware of it.

6.4 Method Lookup and AbCon Vectors

AbCon vectors may be a more efficient mechanism for performing dynamic lookup than method dictionaries, although, to the best of our knowledge, the mechanisms have never been benchmarked side by side. To compare them fairly one must include the time taken to construct the AbCon on assignment. However, for many assignments the compiler is able to determine the concrete type of the object and thus does not need to generate an AbCon, or indeed perform method lookup: the appropriate method can be chosen statically. For many of the remaining assignments, e.g., those where the abstract types of the right- and left-hand sides are the same, the extra cost involved in the assignment is merely the copying of a reference to the AbCon. For some of the remaining assignments, the compiler can determine the Ab-

Con to be built; in these cases the AbCon is built at load time and a reference to it is inserted into the code, so the overhead is reduced to the store of a single constant. For the remaining assignments, the asymptotic cost is reduced by caching AbCons; after a program has run for a while, *all* of the AbCons that it needs will have been cached,¹⁶ but some time is still expended in accessing the cache. In a similar way, the cost of method lookup is normally reduced by caching recent hits. Indeed, polymorphic inline caches [52] have proven so successful in eliminating message lookup that its cost is widely perceived as a non-problem.

6.5 What Made Emerald Fast?

As we discussed in the Introduction, Emerald grew out of our experience with Eden, which had lots of good ideas but rather disappointing performance. One of our major goals was not just to improve on Eden’s performance, but to actually have good performance in an absolute sense (see Section 2). Because performance is hard to retrofit, we practiced performance-oriented design from the beginning. There was no silver bullet: we had to get the details right all along the line.

6.5.1 Single Address Space

A major design decision behind Emerald’s excellent performance was placing all node-local objects and their processes in a single address space (a UNIX process, as described in Section 4.10), which was also shared with the Emerald kernel. This meant that kernel calls were simply procedure calls and any data structures in the object could be accessed by the kernel simply by dereferencing a pointer.

6.5.2 Distributed Operations and Networking

The first Eden remote invocation took approximately 1 second, during which time about 27 messages were sent (counting both IPC messages and Ethernet messages). There was considerable room for improvement: the Eden invocation module was rewritten several times, resulting in substantially fewer messages and a corresponding drop in invocation times, which ended up at approximately 300 ms for a remote invocation and 140 ms for a local invocation. A major lesson from these rewrites was that performance was more or less proportional to the number of messages sent. The best case for node-local invocations in Eden was four IPC messages: one from the source object to the Eden kernel, one from the Eden kernel to the target object, and two more to obtain the result. In Emerald, putting all node-local objects and the kernel into the same address space eliminated IPC messages entirely. The result was that Emerald node-local invocations were more than three orders of magnitude faster than Eden’s, see Section 6.6.

¹⁶We were surprised at how few AbCons even large programs needed.

6.5.3 Choosing Between Eager and Lazy Evaluation

In theory, lazy evaluation is wonderful: nothing is ever evaluated until it is needed, but once evaluated, it is never evaluated again. However, in practice there is a cost to laziness: before a value is used, one must check to see if it is a thunk that first needs to be evaluated. In consequence, eager evaluation is a win in many situations: if it is very likely that a value will be needed, it makes sense to evaluate it early. Moreover, if “early” can be made to mean “at compile time”, then the cost of run-time evaluation can be eliminated entirely.

Eager Evaluation. We applied the idea of aggressive eager evaluation in several places. Using direct references rather than *OIDs* was one of these. In Eden, objects were referred to by a globally unique identifier, which meant that any operation on an object had to translate that global identifier into a reference to some local data structure. In contrast, in Emerald we translated *OIDs* to pointers as soon as possible, so that, for example, a local object was represented by a direct pointer to its data area. Code, AbCons, Types — indeed, anything referenced by *OID* — was represented as a pointer to either the relevant data structure or, in the case of a remote object, to a descriptor that contained information on how to find the data structure (see Section 4.10.1). This made local operations faster, although it also made remote operations slower, because all direct pointers had to be translated when they crossed from one machine to another. However, the slowdown caused by translation was small compared to the cost of a network operation. The principle was that users should pay for a facility only when they used it: we did not want to slow down local operations just to make remote operations faster.

We also avoided lazy evaluation by changing the code to make sure that expensive computations were performed only once. Our experience with Eden had showed that many computations were done repeatedly because several different modules in the implementation needed the result of a single computation. For example, in the early days of Eden invocation, the unique identifier of the invoked object was looked up more than ten times! A quick performance fix was to equip the object table lookup function with a one-item cache, which eliminated the work of the lookup but added the overhead of the cache check. In implementing Emerald we used profiling techniques to discover such inefficiencies and fix them, usually by passing on pointers to resolved data rather than the original *OID* or pointer; this eliminated not only the work of the lookup, but also the overhead of the function call and the cache check.

The ultimate application of eagerness was to move as much computation as possible into the compiler. For example, the compiler did an excellent job of figuring out when it could make an object local rather than global, thus eliminating all the overhead associated with distribution. It also removed as

much of the type information as it could, so if you wrote and compiled a program that did not use distribution at all, then all distribution overhead would be removed from the compiled code. Consequently the program would run at a speed comparable to that of compiled C. We found that on the SPARC architecture, the Emerald version of a highly recursive computation such as the Ackermann function actually ran faster than C, because our calling sequences were more efficient.

Lazy Evaluation. Lazy evaluation was more efficient than aggressive evaluation when the work that was delayed might never need doing at all. A prime example is representing global objects. Global objects were allocated directly by compiled code, but most of the work of making them usable in a distributed environment was delayed until the first time the object had any interaction with another node, for example, by a reference to it being passed to another node. At that time the object would be “baptized” by being given an *OID* and entered into the local object table. Many objects had the *potential* to become known outside the node on which they were created, but most would never actually do so. Thus, postponing baptism was an excellent idea because it was often unnecessary.

Another place where Emerald takes a lazy approach is in moving the code for a migrating object. When an object moves from one node to another, its code is *not* moved along with its data. The reasoning behind this is that the receiving node may well already have a copy of the code, in which case the work would be unnecessary. Of course, if the receiving node does *not* have the code, it has to obtain it; this is actually more work than if the code had been sent eagerly. However, it turns out that most applications have a small working set of code objects, and so most of the time the destination node will already have the code for an immigrating object. In the case where the code does have to be requested, multiple code objects can be requested at the same time, so we win again from a batching effect.

Another technique that used the idea of laziness is replacing computation by compiler-generated tables. One simple example is making the current source-code line number available when debugging. The Simula 67 compiler generated a load instruction that put the current line number into a register. This seemed to us to be optimizing for the uncommon case: the Emerald implementation instead generated a table in each code object that could translate the relative address of any instruction into the line number in the Emerald source code. This had a storage cost but no cost in execution time. Of course, if the debugger was actually activated, it took more time to find the line number, but that was a rare event, and not a time-critical one.

Compiler-generated tables were also used for unavailable and failure handlers. When a failure occurred, such as invoking **nil** or dividing by zero, the appropriate handler was

found by using the address of the failure to search a table of handlers. The implementation of Mesa used the same idea [64]. This was in contrast with other language implementations (for example, Sherman’s Ada compiler for the VAX [95]) where the appropriate failure handler was pushed onto the stack; this made finding the handler faster, but slowed down every call that did not fail.

Lazy evaluation was also the technique of choice for most of the work associated with mobility, because most objects did not move; several examples are described in Section 4.7.3.

6.6 Some Historical Performance Data

From the start, performance was important to the Emerald project; good performance was high on our list of goals, and we hoped that good performance would distinguish our work from contemporary efforts. Of course, by modern standards the performance of any 1980s computer system will be unimpressive, so we summarize here not only Emerald’s performance but also that of some comparable systems available to us at the time.

Most of the performance figures given below were measured in February 1987 on a set of five MicroVAX II machines running Ultrix. The figures in Section 6.6.6 were measured on VAXStation 2000 machines, which had the same CPU as the MicroVAX II but had a different architecture that resulted in slightly longer execution times (usually about 6–8%).

6.6.1 Remote Invocations

As shown in Table 1, remote invocation of a parameterless remote operation took 27.9 ms. This included sending a request message of 160 bytes (i.e., one Ethernet payload) of which 72 bytes were transport-layer protocol headers, 64 bytes were the invocation request and 24 bytes were the return address information. The return message consisted of 82 bytes of which 72 bytes were again transport-layer headers and 12 bytes were the invocation reply.

We measured the low-level transport protocol by sending a 160-byte message and returning a 72-byte message. Such an exchange took 24.5 ms. This means that the actual handling of the remote invocation used only 3.4 ms (12%) of the 27.9 ms.

For comparison, a remote invocation in the Eden system took 54 ms on a SUN 2 (and 140 ms on a VAX 11/750). The major difference between the Eden implementation and Emerald is that Emerald used only two network messages to perform a remote invocation while Eden used four. Bennett’s Distributed Smalltalk implementation used 136 ms for an “empty” remote invocation [11].

Table 1: Remote Operation Timing (MicroVAX II)

Operation	Time/ms
local invocation	0.019
elapsed time, remote invocation	27.9
underlying message exchange	24.5
invocation handling time	3.4

Table 2: Object Creation Timing

Creation of	Global/ms	Δ/ms	Local/ms	Δ/ms
empty object	1.06	—	0.76	—
with an initially	1.34	+0.28	0.92	+0.16
with a monitor	1.34	+0.28	0.96	+0.20
with one <i>Integer</i> variable	1.36	+0.30	0.96	+0.20
with one <i>Any</i> variable	1.37	+0.31	0.96	+0.20
with 100 <i>Integer</i> variables	2.41	+1.26	2.01	+1.25
with 100 <i>Any</i> variables	3.49	+2.43	3.07	+2.31
with one process	2.17	+1.11	1.85	+1.09

Table 3: Remote Parameter Timing

Operation Type	Time/ms	Δ/ms
remote invocation, no parameter	30.3	—
remote invocation, one integer parameter	31.5	+ 1.2
remote invocation, local reference parameter	32.5	+ 2.2
remote invocation, two local reference parameters	34.7	+ 4.4
remote invocation, call-by-move parameter	35.9	+ 5.6
remote invocation, call-by-visit parameter	40.3	+10.0
remote invocation, with one call-back parameter	63.5	+30.2

6.6.2 Object Mobility

Moving a simple data object took about 12 ms. This time is less than the round-trip message time because reply messages are piggybacked on other messages.

6.6.3 Process Mobility

We measured the time taken to move an object containing a small process with six variables. This took 40 ms or about 43% more than the simplest remote invocation. This included sending a message consisting of about 600 bytes of information, including object references, immediate data for replicated objects, a stack segment, and process-control information. The process control information and stack segment together consumed about 180 bytes.

6.6.4 Object Creation

Table 2 shows creation times for various global and local Emerald objects. The numbers were obtained by repeated timings and are median values—averages do not make sense because the timings varied considerably (up to

+50%) when storage allocation caused paging activity. The numbers are correct to about two digits (an error of 1–2%).

In general, it took about 0.3–0.4 ms longer to create an empty global object than an empty local object because of the additional allocation of an object descriptor. An **initially** construct added another 0.2 ms as did the presence of a monitor (because the monitor caused an implicit **initially** to be added to initialize the monitor). Creating an object containing 100 integer variables cost an extra 1.2 ms.

Creating an object containing 100 *Any* variables took about 2.4 ms longer than an empty object because *Any* variables were 8 bytes long while integers were 4 bytes, so twice as much storage had to be allocated and initialized.

6.6.5 Process Creation

Creating a process took an additional 1.1 ms beyond the time required to create its containing object. We measured two processes that took turns calling a monitor. It took 710 μ s for one process switch, one blocking monitor entry, one

unblocking monitor exit, and two kernel calls. By executing two CPU-bound processes and varying the size of the time slice, we estimated the process switching time to be about $300\ \mu s$ ($\pm 5\%$), including the necessary Ultrixcalls to set a timer.

6.6.6 Additional Costs for Parameters

The additional costs of adding parameters to remote invocations were measured in the fall of 1988 on VAXStation 2000s, which ran slightly slower than MicroVAX IIs.

We compared the incremental cost of call by move and call by visit with the incremental cost of call by object reference. We performed an experiment in which an object on a source node S invoked a remote object R and passed as an argument a reference to an object on S . R then invoked the argument object once. Without call by move, this caused a second remote invocation back to S . When using call by move or call by visit, the remote invocation was avoided because the argument object was moved to R . The timings are shown in Table 3.

Table 4 shows the benefit of call by move for a simple argument object containing only a few variables with a total size of less than 100 bytes. The additional cost of call by move over call by reference is 3.4 ms, while call by visit adds 7.8 ms. The call-by-visit time includes sending the invocation message and the argument object, performing the remote invocation (which invokes its argument), and returning the argument object with the reply. One would expect the call-by-visit time to be approximately twice the call-by-move time. It is actually slightly higher due to the dynamic allocation of data structures to hold the call-by-visit control information. Had the argument not been moved, the incremental cost (of the consequent remote invocation) would have been 31.0 ms. These measurements are a lower bound because the cost of moving an object depends on the complexity of the object and the types of the objects it references. The lesson to take away is that call by visit is worthwhile for small parameters, even if they are called *only once*.

6.6.7 Performance of Local Operations

Table 5 shows the performance of several local operations. Integers and reals were implemented as direct objects. The timings for primitive integer and real operations were exactly the same as for comparable operations in C—which is not surprising given that the instructions generated were the same.

For comparison with procedural languages, a C procedure call¹⁷ took $13.4\ \mu s$, a Concurrent Euclid procedure call took $16.4\ \mu s$, and an Emerald local invocation took $16.6\ \mu s$ (i.e., 23% longer than a C procedure call). Concurrent Euclid and Emerald were slower because they had to make an explicit stack overflow check on each call. C avoided this overhead

¹⁷On a MicroVAX II using the Berkeley portable C compiler.

because UNIX used virtual memory hardware to perform stack overflow checks at no additional per-call cost.

The “resident global invocation” time in Table 5 is for a global object (i.e., one that potentially can move around the network) when invoked by another object resident on the same node. The additional $2.8\ \mu s$ (above the time for a local invocation) represents cost of the potential for distribution: the time was spent checking whether or not the invoked object was resident.

6.7 A Local Procedure Call Benchmark

We used Ackermann’s function as a benchmark program because most of its execution time is due to procedure calls; the only other operations performed are tests against zero and integer decrement. We wrote Ackermann’s function in Emerald and in C. The Emerald version appears below:

```
function Ackermann[n: Integer, m: Integer] → [result: Integer]
  if (m = 0) then
    result ← n + 1
  elseif (n = 0) then
    result ← self.Ackermann[1, m - 1]
  else
    result ← self.Ackermann[
      self.Ackermann[n - 1, m], m - 1]
  end if
end Ackermann
```

The C version was written twice: a straightforward version and a hand-optimized version. The straightforward version was timed when compiled both with and without the C optimizer. We compared execution times for two pairs of parameter values, namely (6,3) and (7,3). Table 6 shows the timings along with the relative difference in execution times normalized to the optimized C version. The Emerald version ran about 50% slower than the C version. When the C optimizer was used, the C timings improved by 12–13%. Careful hand optimization improved the timings for C by an additional 10%.

An analysis of the code generated by the C and Emerald compilers revealed that the Emerald version was slower than the C version for three reasons. First, as mentioned earlier, Emerald invocations were 23% slower than C procedure calls. Second, Emerald’s parameter-passing mechanism was more expensive than C’s because Emerald also transferred type information. Third, in Emerald all variables were initialized.

In 1992, Emerald was ported to the SUN SPARC architecture [75]. The SUN C compiler used the SPARC register window, while the Emerald implementation did not. As a consequence, Emerald invocations on the SPARC were almost 15% faster than C procedure calls.

Table 4: Incremental Cost of Remote Invocation Parameters: elapsed time in addition to a call-by-reference invocation

Parameter Passing Mode	Time/ms
empty remote invocation	30.3
call-by-move	+ 3.4
call-by-visit	+ 7.8
call-by-reference, one call-back	+31.0

Table 5: Local Emerald Invocation Timing — MicroVAX II

Emerald Operation	Example	Time/ μ s
primitive integer invocation	$i \leftarrow i + 23$	0.4
primitive real invocation	$x \leftarrow x + 23.0$	3.4
local invocation	localobject.no-op	16.6
resident global invocation	globalobject.no-op	19.4

Table 6: Ackermann’s Function Benchmark (Time in Seconds)

Version	(6,3)	$\Delta\%$	(7,3)	$\Delta\%$
C hand optimized	3.7	-10%	14.9	-10%
C with optimizer	4.1	0%	16.6	0%
C version	4.6	+12%	18.7	+13%
Emerald version	6.6	+61%	27.7	+67%

7. Summary

Emerald is a research programming language that was developed at the University of Washington from 1983 to 1987, and has subsequently been used extensively in teaching and research at the Universities of Copenhagen, British Columbia, and Arizona, at Digital Equipment Corporation, and at Victoria University of Wellington. It was originally implemented on DEC VAX hardware, later on the Motorola 68000-series, then on a portable virtual machine, then on Sun SPARC and Digital Alpha, and most recently in a heterogeneous setting in which objects run native code but can move between architectures. Emerald is object-based in the sense that it supports objects but not classes or inheritance; its distinguishing feature is support for distribution, which is arguably more complete than that of any other language available even now, more than 20 years later. Along the way, Emerald also made significant strides in type theory, including implementations of what have since become known as F-bounded polymorphism and generic data structures.

The primary problems that Emerald sought to address were the costs of object invocation and the coupling between the way that an object was represented in the programming language and the way that the object was implemented. Contemporary object-based distributed languages, notably Argus and the Eden Programming Language, had two notions of object: “small objects”, which were efficiently supported

within a single address space but could not be accessed remotely, and “large objects”, which were accessible from remote address spaces but were thousands of times more costly. Emerald’s contribution was the realization, obvious in hindsight, that these different implementations need not show through to the source language. Instead, the Emerald language has a single notion of object and several different implementation styles: the most appropriate implementation is selected by the compiler depending on how the object is used.

The idea that the users of an object should not know (or care) about the details of its implementation is of course no more than information hiding, a principle that was well known at the time that we were designing Emerald. In addition to using the principle of information hiding in the compiler (the user didn’t have to know how the compiler implemented a object), we also made it available to the programmer. We did this by thoroughly separating the notions of class (how an object is implemented) from those of type (what interface it presents to other objects). Emerald’s type system is based on the notion of structural type conformity, which rigorously specifies when an implementation of an object satisfies the demands of a type, and also when one type subsumes another.

Letting the compiler choose an implementation was only possible if this did not change the semantics, so we were

forced to define the semantics abstractly. Thus, the semantics of parameter passing cannot depend on the relative locations of the invoker and the invoked. Emerald does include features to control the location of objects: to move an object to a new location, to fix and unfix an object at a specific location, to move an object to a remote location with an invocation, and to cause an object to visit a remote location for the duration of an invocation. Using these features does not change the semantics of a program, but may dramatically change its performance. Emerald also distinguishes functions (which have no effect on the state) from more general operation invocations, and immutable objects (which the implementation can replicate) from mutable objects.

Emerald’s main deficiency, viewed from the vantage-point of today, is its lack of inheritance. Given the absence of classes from the language itself and the decentralized implementation strategy, it was not clear how inheritance could be incorporated. After gaining some experience with the language, and in particular with the tedium of writing two-level object constructors all the time, we did add classes, including single inheritance, as “syntactic sugar” for the obvious two-level object constructor. The resolution of inherited attributes was done entirely at compile time, implying that the text of the superclass had to be available to the compiler when the subclass was compiled. In addition, this simple scheme offered no support for “super” or otherwise invoking inherited methods in the body of subclass methods: a method or instance variable in the subclass completely redefined, rather than just overrode, any identically named method or variable inherited from the superclass. Rather than pursuing more traditional inheritance in the language, subsequent work looked at compile-time code reuse [89].

One issue that the Emerald implementation never addressed was ensuring that a remote object that claimed to be of a certain type did in fact conform to that type. Type information was present in the run-time object structures in two forms: an object structure representing the type, which was used for conformity checking, and the executable code, which actually implemented the operations required by the type. While a correct Emerald compiler always guaranteed that the object structure and the code corresponded, a malicious user on a remote node could in principle hack a version of the compiler to void this guarantee, thus breaching the type security of the whole system. We imagined overcoming this problem by certifying the compiler and having each compiler sign its own code, but this was never implemented.

Although Emerald’s support for remote invocation has been widely reproduced, remote invocation has rarely been implemented with such semantic transparency. Implementations of mobile objects are still rare and languages that incorporate mobility into their semantic model rarer still, although recently proposals have been made for incorporating mobility into Java.

Acknowledgements

This paper has taken shape over a long period, and has been much improved by the contributions of many colleagues. Our HOPL referees, Brent Hailpern, Doug Lee, and Barbara Ryder, all provided useful advice. Our external referee, Andrew Watson, along with Andrew Herbert, helped to reconstruct the influence of the Emerald type system. Kim Bruce, Sacha Krakowiak, Roy Levin, and Jim Waldo helped fill in numerous details, and Michael Mahoney helped us learn how to write history.

We thank the Danish Center for Grid Computing, Microsoft Research Cambridge, the University of Copenhagen, and Portland State University for their generous support while this paper was being written.

References

- [1] J. Mack Adams and Andrew P. Black. On proof rules for monitors. *Operating Systems Review*, 16(2):18–27, April 1982.
- [2] J. E. Allchin and M. S. McKendry. Synchronization and recovery of actions. In *Proceedings of the 2nd Symposium on Principles of Distributed Computing*, pages 31–44, New York, NY, USA, August 1983. SIGOPS/SIGACT, ACM Press.
- [3] G. T. Almes, A. P. Black, E. D. Lazowska, and J. D. Noe. The Eden system: A technical review. *IEEE Transactions on Software Engineering*, SE11(1):43–59, January 1985.
- [4] Guy T. Almes. *Garbage Collection in an Object-Oriented System*. PhD thesis, Carnegie Mellon University, June 1980.
- [5] Allan Thrane Andersen and Ole Høegh Hansen. Teoretiske og praktiske forhold ved brugen af mønstre i forbindelse med udvikling af objektorienteret software-med særlig fokus på sammenhængen mellem mønstre og programmeringssprog. M.Sc. thesis, DIKU, University of Copenhagen, 1999.
- [6] ANSI. *Draft American National Standard for Information Systems—Programming Languages—Smalltalk*. ANSI, December 1997. Revision 1.9.
- [7] Ken Arnold, Bryan O’Sullivan, Robert W. Scheifler, Jim Waldo, and Ann Wollrath. *The Jini Specification*. Addison Wesley, 1999.
- [8] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Report on the algorithmic language Algol 60. *Communications of the ACM*, 3(5):299–314, 1960.
- [9] L. Frank Baum. *The Wonderful Wizard of Oz*. George M. Hill, Chicago, 1900.
- [10] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison Wesley, 2000.
- [11] John K. Bennett. The design and implementation of distributed Smalltalk. In Norman K. Meyrowitz, editor, *Proceedings of the Second ACM Conf. on Object-Oriented*

- Programming Systems, Languages and Applications*, pages 318–330, New York, NY, USA, October 1987. ACM Press. Published as SIGPLAN Notices 22(12), December 1987.
- [12] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. Network objects. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 217–230, Asheville, NC (USA), December 1993. ACM Press.
 - [13] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. Network Objects. Technical Report 115, Digital Systems Research Center, Palo Alto, CA, December 1994.
 - [14] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.
 - [15] Graham M. Birtwistle, Ole-Johan Dahl, Bjorn Myhrtag, and Kristen Nygaard. *Simula BEGIN*. Auerbach Press, Philadelphia, 1973.
 - [16] Andrew Black, Larry Carter, Norman Hutchinson, Eric Jul, and Henry M. Levy. Distribution and abstract types in Emerald. Technical Report 86-02-04, Department of Computer Science, University of Washington, Seattle, February 1986.
 - [17] Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy. Object structure in the Emerald system. In Norman K. Meyrowitz, editor, *Proceedings of the First ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 78–86. ACM, October 1986. Published in SIGPLAN Notices, 21(11), November 1986.
 - [18] Andrew Black, Norman Hutchinson, Eric Jul, Henry Levy, and Larry Carter. Distribution and abstract data types in Emerald. *IEEE Transactions on Software Engineering*, SE-13(1):65–76, January 1987.
 - [19] Andrew Black, Norman Hutchinson, Eric Jul, and Henry M. Levy. Distribution and abstract types in Emerald. Technical Report 85-08-05, Department of Computer Science, University of Washington, Seattle, August 1985.
 - [20] Andrew P. Black. *Exception Handling: the Case Against*. D.Phil thesis, University of Oxford, January 1982. <http://web.cecs.pdx.edu/~black/publications/Black%20D.%20Phil%20Thesis.pdf>.
 - [21] Andrew P. Black. The Eden programming language. Technical Report TR 85-09-01, Department of Computer Science, University of Washington, September 1985.
 - [22] Andrew P. Black. Supporting distributed applications: experience with Eden. In *SOSP '85: Proceedings of the tenth ACM symposium on Operating systems principles*, pages 181–193, New York, NY, USA, 1985. ACM Press.
 - [23] Andrew P. Black. Object identity. In *Proc. of the Third Int'l Workshop on Object Orientation in Operating Systems (IWOOS'93)*, Asheville, NC, December 1993. IEEE Computer Society Press.
 - [24] Andrew P. Black and Yeshauahu Artsy. Implementing location independent invocation. In *Proc. 9th International Conference on Distributed Computing Systems*, pages 550–559. IEEE Computer Society Press, June 1989.
 - [25] Andrew P. Black and Yeshauahu Artsy. Implementing location independent invocation. *IEEE Transactions on Parallel and Distributed Syst.*, 1(1):107–119, 1990.
 - [26] Andrew P. Black and Norman Hutchinson. Typechecking polymorphism in Emerald. Technical Report CRL 91/1, Digital Cambridge Research Laboratory, One Kendall Square, Building 700, Cambridge, MA 02139, December 1990.
 - [27] Andrew P. Black and Mark P. Immel. Encapsulating plurality. In Oscar Nierstrasz, editor, *Proceedings ECOOP '93*, volume 707 of *Lecture Notes in Computer Science*, pages 57–79, Kaiserslautern, Germany, July 1993. Springer-Verlag.
 - [28] Toby Bloom. Immutable groupings. CLU Design Note 61, MIT—Project MAC, August 1976.
 - [29] Per Brinch Hansen. The programming language Concurrent Pascal. *IEEE Transactions on Software Engineering*, 1(2):199–207, June 1975.
 - [30] Per Brinch Hansen. *The Architecture of Concurrent Programming*. Prentice Hall Series in Automatic Computation. Prentice Hall Inc., Englewood Cliffs, New Jersey, 1977.
 - [31] Andrei Z. Broder. Some applications of Rabin's fingerprinting method. In Renato Capocelli, Alfredo De Santis, and Ugo Vaccaro, editors, *Sequences II: Methods in Communications, Security, and Computer Science*, pages 143–152. Springer-Verlag, 1993.
 - [32] Kim B. Bruce. Safe type checking in a statically-typed object-oriented programming language. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 285–298, Charleston, South Carolina, United States, 1993. ACM Press.
 - [33] Kim B. Bruce, Adrian Fiech, and Leaf Petersen. Subtyping is not a good “match” for object oriented languages. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proceedings ECOOP '97*, volume 1241 of *Lecture Notes in Computer Science*, pages 104–127, Jyväskylä, Finland, June 1997. Springer-Verlag.
 - [34] Peter S. Canning, William Cook, Walter L. Hill, John C. Mitchell, and Walter G. Olthoff. F-bounded polymorphism for object-oriented programming. In *Proceedings of the ACM Conference on Functional Programming and Computer Architecture*, pages 273–280, September 1989.
 - [35] Luca Cardelli. A semantics of multiple inheritance. In Gilles Kahn, David B. MacQueen, and Gordon D. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 51–67. Springer, 1984.
 - [36] Luca Cardelli. Structural subtyping and the notion of power type. In *Proceedings of the Fifteenth Symposium on Principles of Programming Languages*, pages 70–79, San Diego, CA, January 1988. ACM.
 - [37] Luca Cardelli. A language with distributed scope. *Computing Systems*, 8(1):27–59, 1995.
 - [38] Luca Cardelli, Jim Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 language

- definition. *ACM SIGPLAN Notices*, 27(8):15–42, August 1992.
- [39] Luca Cardelli, Jim Donahue, Mick Jordan, Bill Kalsow, and Greg Nelson. The Modula-3 type system. In *Proceedings of the Sixteenth Symposium on Principles of Programming Languages*, pages 202–212, Austin, Texas, United States, January 1989. ACM Press.
- [40] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.
- [41] Ole-Johan Dahl, Bjørn Myhrhaug, and Kristen Nygaard. Common base language. Technical Report S-22, Norwegian Computing Center, October 1970.
- [42] Wolfgang De Meuter. *Move Considered Harmful: A Language Design Approach to Mobility and Distribution for Open Networks*. PhD thesis, Vrije Universiteit Brussel, September 2004.
- [43] A. Demers and J. Donahue. Revised report on Russell. Technical Report TR 79-389, Computer Science Department, Cornell University, 1979.
- [44] James Donahue and Alan Demers. Data types are values. *ACM Transactions on Programming Languages and Systems*, 7(3):426–445, July 1985.
- [45] Bradley M. Duska. Enforcing crash failure semantics in distributed systems with fine-grained object mobility. M.Sc. thesis, Computer Science Department, University of British Columbia, August 1998.
- [46] International Organization for Standardization. Information technology—open distributed processing—reference model. International Standard 10746, ISO/IEC, 1998.
- [47] Arthur Neal Glew. Type systems in object oriented programming languages. B.sc. (hons) thesis, Victoria University of Wellington, 1993.
- [48] Xiaomei Han. Memory reclamation in Emerald—an object-oriented programming language. M.Sc. thesis, Computer Science Department, University of British Columbia, June 1994.
- [49] C.A.R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.
- [50] R. C. Holt. A short introduction to Concurrent Euclid. *SIGPLAN Notices*, 17(5):60–79, May 1982.
- [51] R. C. Holt. *Concurrent Euclid, the UNIX System, and TUNIS*. Addison-Wesley Publishing Company, 1983.
- [52] Urs Hözle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In P. America, editor, *Proceedings ECOOP '91*, volume 512 of *Lecture Notes in Computer Science*, pages 21–38, Geneva, Switzerland, July 1991. Springer-Verlag.
- [53] Norm Hutchinson and Eric Jul. The handling of unix signals in the concurrent euclid kernel. Eden memo, Dept. of Computer Science, University of Washington, 1984.
- [54] Christian Damsgaard Jensen. Fine-grained object based load distribution—an experiment with load distribution in guide-2. M.Sc. thesis, DIKU, University of Copenhagen, 1995.
- [55] Lars Rye Jeppesen and Søren Frøkjær Thomsen. Generational, distributed garbage collection for emerald. M.Sc. thesis, DIKU, University of Copenhagen, 2002.
- [56] Nick Jørding and Flemming Stig Andreasen. A distributed wide area name service for an object oriented programming system. M.Sc. thesis, DIKU, University of Copenhagen, 1994.
- [57] Eric Jul. Structuring of dedicated concurrent programs using adaptable I/O interfaces. Master’s thesis, DIKU, Department of Computer Science, University of Copenhagen, Copenhagen, Denmark, December 1980. Technical Report no. 82/3.
- [58] Eric Jul. *Object Mobility in a Distributed Object-Oriented System*. Ph.D. thesis, Department of Computer Science, University of Washington, Seattle, December 1988.
- [59] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.
- [60] Niels Christian Juul. *Comprehensive, Concurrent and Robust Garbage Collection in the Distributed, Object-Based System Emerald*. PhD thesis, DIKU, Department of Computer Science, University of Copenhagen, February 1993. DIKU Report 93/1.
- [61] Niels Christian Juul and Eric Jul. Comprehensive and robust garbage collection in a distributed system. In *International Workshop on Memory Management (IWMM)*, volume 637 of *Lecture Notes in Computer Science*, pages 103–115, 1992.
- [62] Setrag N. Khoshafian and George P. Copeland. Object identity. In Norman K. Meyrowitz, editor, *Proceedings of the First ACM Conf. on Object-Oriented Programming Systems, Languages and Applications*, pages 406–416, Portland, Oregon, October 1986. Published as SIGPLAN Notices 21(11), November 1986.
- [63] Jan Kølander. Implementation af osi-protokoller i det distribuerede system emerald vha. isode. M.Sc. thesis, DIKU, University of Copenhagen, 1994.
- [64] B. W. Lampson, J. G. Mitchell, and E. H. Satterthwaite. On the transfer of control between contexts. In *Lecture Notes in Computer Science: Programming Symposium*, volume 19, pages 181–203. Springer-Verlag, 1974.
- [65] Niels Elgaard Larsen. An object-oriented database in emerald. M.Sc. thesis, DIKU, University of Copenhagen, 1992.
- [66] Niels Elgaard Larsen. *Emerald Database—Integrating Transaction, Queries, and Method Indexing into a system based on Mobile Objects*. PhD thesis, DIKU, Department of Computer Science, University of Copenhagen, February 2006.
- [67] Edward D. Lazowska, Henry M. Levy, Guy T. Almes,

- Michael J. Fischer, Robert J. Fowler, and Stephen C. Vestal. The architecture of the Eden system. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, pages 148–159, Pacific Grove, California, 1981. ACM Press.
- [68] Morten Jes Lehrmann. Load distribution in Emerald—an experiment. M.Sc. thesis, DIKU, University of Copenhagen, 1994.
- [69] Hank Levy, Norm Hutchinson, and Eric Jul. Getting to Oz, April 1984. Included as an appendix to this article.
- [70] Henry M. Levy. *Capability-Based Computer Systems*. Digital Press, 1984.
- [71] Barbara Liskov. Distributed programming in Argus. *Communications of the ACM*, 31(3):300–312, March 1988.
- [72] Barbara Liskov. A history of CLU. In Thomas A. Bergin and Richard G. Gibson, editors, *History of Programming Languages*, chapter 10, pages 471–510. ACM Press, New York, NY, USA, 1996.
- [73] Barbara Liskov, Alan Snyder, Robert Atkinson, and Craig Schaffert. Abstraction mechanisms in CLU. *Communications of the ACM*, 20(8):564–576, August 1977.
- [74] Gus Lopez, Bjorn Freeman-Benson, and Alan Borning. Constraints and object identity. In M. Tokoro and R. Pareschi, editors, *Proceedings ECOOP '94*, volume 821 of *Lecture Notes in Computer Science*, pages 260–279, Bologna, Italy, July 1994. Springer-Verlag.
- [75] Jacob Marquardt. Porting emerald to a sparc. M.Sc. thesis, DIKU, University of Copenhagen, 1992.
- [76] Robert M. Metcalfe and David R. Boggs. Ethernet: distributed packet switching for local computer networks. *Communications of the ACM*, 19(7):395–404, July 1976.
- [77] Albert R. Meyer and Mark B. Reinhold. ‘Type’ is not a type: Preliminary report. In *Proceedings of the Thirteenth Symposium on Principles of Programming Languages*, pages 287–295, St. Petersburg, January 1986. ACM.
- [78] Jeppe Damkjær Nielsen. Paradigms for the design of distributed operating systems. M.Sc. thesis, DIKU, University of Copenhagen, 1995.
- [79] E. Organick. *A Programmer’s View of the Intel 432 System*. McGraw-Hill, 1983.
- [80] Dimokritos Michael Papadopoulos and Kenneth Folmer-Petersen. Porting the LAN-based distributed system Emerald to a WAN. Master’s thesis, DIKU, Department of Computer Science, University of Copenhagen, Copenhagen, Denmark, 1993.
- [81] P. Pardyak. Group communication in a distributed object-based system. Master’s thesis, University of Mining and Metallurgy, Krakow, Poland, September 1992. Technical Report TR-92-1.
- [82] P. Pardyak. Group communication in an object-based environment. In *1992 Int. Workshop on Object Orientation in Operating Systems*, pages 106–116, Dourdan, France, 1992. IEEE Computer Society Press.
- [83] P. Pardyak and B. Bershad. A group structuring mechanism for a distributed object-oriented language. In *Proc. of the 14th International Conf. on Distributed Computing Systems*, pages 312–319. IEEE Computer Society Press, July 1994.
- [84] Przemysław Pardyak and Krzysztof Zielinski. Emerald—język i system rozproszonego programowania obiektowego (Emerald—a language and system for distributed object-oriented programming). In *Srodowiska Programowania Rozprozonego w Sieciach Komputerowych, (Distributed Programming Environments in Computer Networks)*. Księgarnia Akademicka, Krakow, 1994.
- [85] David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.
- [86] Margaret A. S. Petrus. Service migration in a gigabit network. M.Sc. thesis, Computer Science Department, University of British Columbia, August 1998.
- [87] Simon Pohlen. Maintainable concurrent software. M.Sc. thesis, Victoria University of Wellington, April 1997.
- [88] Michael O. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.
- [89] Rajendra K. Raj and Henry M. Levy. A compositional model for software reuse. In S. Cook, editor, *Proceedings ECOOP ’89*, pages 3–24, Nottingham, July 1989. Cambridge University Press.
- [90] Rajendra K. Raj, Ewan Tempero, Henry M. Levy, Andrew P. Black, Norman C. Hutchinson, and Eric Jul. Emerald: a general-purpose programming language. *Software—Practice and Experience*, 21(1):91–118, 1991.
- [91] Rajendra Krishna Raj. *Composition and reuse in object-oriented languages*. PhD thesis, Department of Computer Science, University of Washington, Seattle, WA, USA, 1991.
- [92] Craig Schaffert. Immutable groupings. CLU Design Note 47, MIT—Project MAC, April 1975.
- [93] Bodil Schrøder. Mik—et korutineorienteret styresystem til en mikrodatamat. DIKU Blue Report 76/1, DIKU, Dept. of Computer Science, 1976.
- [94] Marc Shapiro, Yvon Gourhant, Sabine Habert, Laurence Mosseri, Michel Ruffin, and Céline Valot. SOS: An object-oriented operating system—assessment and perspectives. *Computing Systems*, 2(4):287–338, December 1989.
- [95] Mark Sherman, Andy Hisgen, David Alex Lamb, and Jonathan Rosenberg. An Ada code generator for VAX 11/780 with Unix. In *Proceeding of the ACM-SIGPLAN symposium on Ada programming language*, pages 91–100, Boston, Massachusetts, 1980. ACM Press.
- [96] Sys Sidenius and Eric Jul. K8080—Flytning af Concurrent Pascal til Intel 8080. Technical Report 79/9, DIKU, Dept. of Computer Science, University of Copenhagen, 1979.
- [97] Bjarne Steensgaard and Eric Jul. Object and native code thread mobility among heterogeneous computers. In *SOSP*, pages 68–78, 1995.
- [98] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey*

- Approach to Programming Language Theory.* MIT Press, 1977.
- [99] Robert E. Strom, David F. Bacon, Arthur P. Goldberg, Andy Lowry, Daniel M. Yellin, and Shaula Alexander Yemini. *Hermes: A Language for Distributed Computing*. Prentice Hall, 1991.
 - [100] Bjarne Stroustrup. The history of C++: 1979–1991. In *Proc. ACM History of Programming Languages Conference (HOPL-2)*. ACM Press, March 1993.
 - [101] Andrew Tanenbaum. *Computer Networks*. Prentice-Hall, first edition, 1980.
 - [102] David M. Ungar. *The Design And Evaluation Of A High Performance SMALLTALK System*. PhD thesis, University of California at Berkeley, Berkeley, CA, USA, 1986.
 - [103] Adriaan van Wijngaarden, B.J. Mailloux, J.E.L. Peck, C.H.A. Koster, M. Sintoff, C.H. Lindsey, L.G.L.T. Meertens, and R.G. Fisker. *Revised Report on the Algorithmic Language Algol 68*. Springer Verlag, 1976.
 - [104] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. A note on distributed computing. Technical Report SMLI TR-94-29, Sun Microsystems Labs, November 1994.
 - [105] Timothy Walsh. *The Taxy Mobility System*. PhD thesis, Trinity College Dublin, June 2006.
 - [106] Andrew J. Watson. Advanced networked systems architecture — an application programmer’s introduction to the architecture. Technical Report TR.017.00, APM Ltd., November 1991.
 - [107] Wikipedia. Duck typing, September 2006. http://en.wikipedia.org/wiki/Duck_typing.
 - [108] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. Hydra: the kernel of a multiprocessor operating system. *Communications of the ACM*, 17(6):337–345, June 1974.

A. Appendix: Historical Documents

The *Getting to Oz* document is reproduced on the following pages. This and other historical documents are available at <http://www.emeraldprogramminglanguage.org>.

Getting to Oz

Hank Levy, Norm Hutchinson, Eric Jul

April 27, 1984

For the past several months, the three of us have been discussing the possibility of building a new object-based system. Carl Binding frequently attended our meetings, and occasionally Guy Almes, Andrew Black, or Alan Borning sat in also. This system, called Oz, is an outgrowth of our experience with the Eden system. In this memo we try to capture the background that led to our current thinking on Oz. This memo is not a specification but a brief summary of the issues discussed in our meetings.

Starting in winter term, 1984, we began to examine some of the strengths and weaknesses of Eden. Several of the senior Eden graduate students had been experimenting with improving Eden's performance. Although they were able to significantly decrease Eden invocation costs, performance was still far from acceptable. Certainly some of the performance problem was due to Eden's current invocation semantics, some was due to implementation of invocation, and some was due to the fact that Eden is built on top of the Unix system.

In addition to performance problems, Eden suffered from the lack of a clean interface. That is, the Eden programmer needs to know about Eden, about EPL (Eden Programming Language -- an preprocessor-implemented extension to Concurrent Euclid), and about Unix to build Eden applications. Also, there was at that time no Eden user interface. Users built Eden applications with the standard Unix command system.

This combination of issues led us to consider building a better integrated system from scratch. Performance was at the top of our priorities. To date, object systems have a reputation of being slow and we don't think this is inherently due to their support for objects. We want to build a distributed object-based system with performance comparable to good message passing systems. To do this, we would have to build a low-level, bare-machine kernel and compiler support. In addition, we would like our system to have an object-based user interface as well as an object-based programming interface. Thus, users should be able to create and manipulate objects from a display.

Our first discussions concentrated on low-level kernel issues. In the Eden system, there are two types of processes and two levels of scheduling. Applications written in EPL contain multiple lightweight processes that coexist within a single Unix address space. These processes are scheduled by a kernel running within that address space. This kernel gains control through special checks compiled into the application. At the next level, multiple address spaces (Unix processes) are scheduled by the Unix system.

Our first decision was that our system would provide both lightweight processes that share

an address space and multiple address spaces. Multiple address spaces would add protection, particularly between multiple languages if they were supported. Within a single address space the compiler would provide protection. A single scheduler within Oz would schedule both light and heavy processes, making the obvious tradeoffs to schedule lightweight processes within the same address space if possible. This would remove the overhead currently caused by the two-level scheduler in Eden.

We wanted lightweight processes for two reasons. First, switching between them involves much less work and is therefore more efficient. Second, and more important, we wanted to reduce the cost of many invocations to that of a procedure call or less. This requires that objects share an address space and leads to the next issue -- compiler support.

Thus, a major goal of Oz is exploitation of compiler technology to increase system performance. In Eden, EPL generates calls to routines that package invocation parameters into a standard interchange format that must be type-checked at the receiving end. The compiler for Oz would instead perform compile-time type checking and produce efficient code for invocation. In addition, the Oz compiler would examine the use of objects invoked from a particular object and would generate code according to that use. For example, if the compiler discovered that object A's use was strictly local to object B, it could produce inline code within B for the manipulation of A's representation.

It is interesting that up to this point our approach had been mostly from the kernel level. We had discussed address spaces, sharing, local and remote invocation, and scheduling. However, we began to realize more and more that the kernel issues were not at the heart of the project. Eventually, we all agreed that language design was the fundamental issue. Our kernel is just a run-time system for the Oz language (called Toto) and the interesting questions were the semantics supported by Toto. We also had spent much time discussing the ways in which Eden is and is not object based. That is, Eden is object based in the sense of Hydra but not in the sense of CLU or Smalltalk.

One thinks in terms of objects when designing distributed applications on Eden. In fact, we all agreed that Eden had been a success in demonstrating the ease with which distributed applications could be programmed using location-independent objects. However, the implementation of Eden objects as large entities restricts their use to certain classes of resources. For smaller data abstractions, one must drop into EPL and use its type facilities. These type facilities require different abstractions to define things that are essentially objects. In other words, Eden supports object-based programming in the large but not in the small. This is particularly troublesome if one believes, as we do, that most objects are local to the application and will never be distributed. Why then should they pay the cost?

Another goal, then, and a difficult one, is to design a language that supports both large objects (typically, operating system resources such as files, mailboxes, etc.) and small objects (typically, data abstractions such as queues, etc.) using a *single* semantics. Large objects might contain processes, be shared, and move from node to node. Small objects are used within other objects to implement simple abstractions. They are not shared or moved. They

typically contain data only. Within Toto, all objects are defined the same way, however the compiler implements objects differently depending on their usage. In this way, we use invocation mechanisms whose performance is commensurate with the needs of the object. For example, different objects may be invoked through direct inline code, through procedure call, or through message passing.

Oz is intended to run on a set of homogeneous nodes on a local area network. Since we're experimenting with a distributed system, we decided that the concept of object location should be supported by the language. The movement of objects should be easily expressible in the language. In fact, this is an advantage of object-based programming that we wish to demonstrate. Other systems provide message passing or even process migration but have no clean concept of *what* it is that is moved. In Oz, objects are the unit of movement and an object can consist of data, a process, or both. The language includes statements that (1) determine the location of an object, (2) move an object to a particular site, and (3) fix an object at a particular site.

We are not changing the Eden concept of location-independent invocation. Rather, we say in Oz that invocation is location-independent but objects are not. That is, location is a fundamental part of each object's state. An Oz program can locate and move other objects. A load balancing object may locate and move objects based on system usage information. However, the semantics of object invocation are identical for local and remote objects.

We have recently discussed implementation strategies for invocation parameter passing. Parameter passing is made difficult by the distributed nature of the system and by the options in object size and processing (i.e., whether an object has a process or not). In general, invocation parameters are passed by reference. Some small immutable objects may be passed by value for performance reasons; for example, we would not pass a reference to the integer 3. A new parameter passing mechanism we've considered is *call by move*, in which the invoked operation explicitly indicates that the parameter object should be moved to the location of the invoked object. This could, in fact, be the principal facility for object movement in Oz.

Finally, because we have a distributed system, the language must allow the programmer to deal with failures. In Oz, failures are *anticipated* conditions that arise due to the distributed nature of the system. For example, applications and nodes go up and down, links break, etc. Programs should be written to expect and deal with these failures. Program bugs, on the other hand, such as array out of bounds violations or divide by zero, are not handled in Oz. We don't intend to provide full-blown exception handling for these conditions.

To summarize, we are considering building an *object-based language* that supports:

1. objects (in particular, both large and small objects are supported by a single language semantics)
2. efficient invocation (both local and remote, both within an address space and outside an address space), and

3. distribution (objects have location and are potentially movable).

Except for these three issues, the rest of the language will be extremely simple. We are currently working on a preliminary language spec which will be available shortly.

A History of Haskell: Being Lazy With Class

Paul Hudak

Yale University

paul.hudak@yale.edu

John Hughes

Chalmers University

rjmh@cs.chalmers.se

Simon Peyton Jones

Microsoft Research

simonpj@microsoft.com

Philip Wadler

University of Edinburgh

wadler@inf.ed.ac.uk

Abstract

This paper describes the history of Haskell, including its genesis and principles, technical contributions, implementations and tools, and applications and impact.

1. Introduction

In September of 1987 a meeting was held at the conference on Functional Programming Languages and Computer Architecture in Portland, Oregon, to discuss an unfortunate situation in the functional programming community: there had come into being more than a dozen non-strict, purely functional programming languages, all similar in expressive power and semantic underpinnings. There was a strong consensus at this meeting that more widespread use of this class of functional languages was being hampered by the lack of a common language. It was decided that a committee should be formed to design such a language, providing faster communication of new ideas, a stable foundation for real applications development, and a vehicle through which others would be encouraged to use functional languages.

These opening words in the Preface of the first Haskell Report, Version 1.0 dated 1 April 1990, say quite a bit about the history of Haskell. They establish the motivation for designing Haskell (the need for a common language), the nature of the language to be designed (non-strict, purely functional), and the process by which it was to be designed (by committee).

Part I of this paper describes genesis and principles: how Haskell came to be. We describe the developments leading up to Haskell and its early history (Section 2) and the processes and principles that guided its evolution (Section 3).

Part II describes Haskell's technical contributions: what Haskell is. We pay particular attention to aspects of the language and its evo-

lution that are distinctive in themselves, or that developed in unexpected or surprising ways. We reflect on five areas: syntax (Section 4); algebraic data types (Section 5); the type system, and type classes in particular (Section 6); monads and input/output (Section 7); and support for programming in the large, such as modules and packages, and the foreign-function interface (Section 8).

Part III describes implementations and tools: what has been built for the users of Haskell. We describe the various implementations of Haskell, including GHC, hbc, hugs, nhc, and Yale Haskell (Section 9), and tools for profiling and debugging (Section 10).

Part IV describes applications and impact: what has been built by the users of Haskell. The language has been used for a bewildering variety of applications, and in Section 11 we reflect on the distinctive aspects of some of these applications, so far as we can discern them. We conclude with a section that assesses the impact of Haskell on various communities of users, such as education, open-source, companies, and other language designers (Section 12).

Our goal throughout is to tell the story, including who was involved and what inspired them: the paper is supposed to be a *history* rather than a technical description or a tutorial.

We have tried to describe the evolution of Haskell in an even-handed way, but we have also sought to convey some of the excitement and enthusiasm of the process by including anecdotes and personal reflections. Inevitably, this desire for vividness means that our account will be skewed towards the meetings and conversations in which we personally participated. However, we are conscious that many, many people have contributed to Haskell. The size and quality of the Haskell community, its breadth and its depth, are both the indicator of Haskell's success and its cause.

One inevitable shortcoming is a lack of comprehensiveness. Haskell is now more than 15 years old and has been a seedbed for an immense amount of creative energy. We cannot hope to do justice to all of it here, but we take this opportunity to salute all those who have contributed to what has turned out to be a wild ride.

©2007 ACM 978-1-59593-766-7/2007/06-ART12 \$5.00

DOI 10.1145/1238844.1238856

<http://doi.acm.org/10.1145/1238844.1238856>

Part I

Genesis and Principles

2. The genesis of Haskell

In 1978 John Backus delivered his Turing Award lecture, “Can programming be liberated from the von Neumann style?” (Backus, 1978a), which positioned functional programming as a radical attack on the whole programming enterprise, from hardware architecture upwards. This prominent endorsement from a giant in the field—Backus led the team that developed Fortran, and invented Backus Naur Form (BNF)—put functional programming on the map in a new way, as a practical programming tool rather than a mathematical curiosity.

Even at that stage, functional programming languages had a long history, beginning with John McCarthy’s invention of Lisp in the late 1950s (McCarthy, 1960). In the 1960s, Peter Landin and Christopher Strachey identified the fundamental importance of the lambda calculus for modelling programming languages and laid the foundations of both operational semantics, through abstract machines (Landin, 1964), and denotational semantics (Strachey, 1964). A few years later Strachey’s collaboration with Dana Scott put denotational semantics on firm mathematical foundations underpinned by Scott’s domain theory (Scott and Strachey, 1971; Scott, 1976). In the early ’70s, Rod Burstall and John Darlington were doing program transformation in a first-order functional language with function definition by pattern matching (Burstall and Darlington, 1977). Over the same period David Turner, a former student of Strachey, developed SASL (Turner, 1976), a pure higher-order functional language with lexically scoped variables—a sugared lambda calculus derived from the applicative subset of Landin’s ISWIM (Landin, 1966)—that incorporated Burstall and Darlington’s ideas on pattern matching into an executable programming language.

In the late ’70s, Gerry Sussman and Guy Steele developed Scheme, a dialect of Lisp that adhered more closely to the lambda calculus by implementing lexical scoping (Sussman and Steele, 1975; Steele, 1978). At more or less the same time, Robin Milner invented ML as a meta-language for the theorem prover LCF at Edinburgh (Gordon et al., 1979). Milner’s polymorphic type system for ML would prove to be particularly influential (Milner, 1978; Damas and Milner, 1982). Both Scheme and ML were strict (call-by-value) languages and, although they contained imperative features, they did much to promote the functional programming style and in particular the use of higher-order functions.

2.1 The call of laziness

Then, in the late ’70s and early ’80s, something new happened. A series of seminal publications ignited an explosion of interest in the idea of *lazy* (or non-strict, or call-by-need) functional languages as a vehicle for writing serious programs. Lazy evaluation appears to have been invented independently three times.

- Dan Friedman and David Wise (both at Indiana) published “Cons should not evaluate its arguments” (Friedman and Wise, 1976), which took on lazy evaluation from a Lisp perspective.
- Peter Henderson (at Newcastle) and James H. Morris Jr. (at Xerox PARC) published “A lazy evaluator” (Henderson and Morris, 1976). They cite Vuillemin (Vuillemin, 1974) and Wadsworth (Wadsworth, 1971) as responsible for the origins of the idea, but popularised the idea in POPL and made one other important contribution, the name. They also used a variant of

Lisp, and showed soundness of their evaluator with respect to a denotational semantics.

- David Turner (at St. Andrews and Kent) introduced a series of influential languages: SASL (St Andrews Static Language) (Turner, 1976), which was initially designed as a strict language in 1972 but became lazy in 1976, and KRC (Kent Recursive Calculator) (Turner, 1982). Turner showed the elegance of programming with lazy evaluation, and in particular the use of lazy lists to emulate many kinds of behaviours (Turner, 1981; Turner, 1982). SASL was even used at Burroughs to develop an entire operating system—almost certainly the first exercise of pure, lazy, functional programming “in the large”.

At the same time, there was a symbiotic effort on exciting new ways to *implement* lazy languages. In particular:

- In software, a variety of techniques based on *graph reduction* were being explored, and in particular Turner’s inspirationally elegant use of *SK combinators* (Turner, 1979b; Turner, 1979a). (Turner’s work was based on Haskell Curry’s *combinatory calculus* (Curry and Feys, 1958), a variable-less version of Alonzo Church’s lambda calculus (Church, 1941).)
- Another potent ingredient was the possibility that all this would lead to a radically different non-von Neumann hardware architectures. Several serious projects were underway (or were getting underway) to build *dataflow* and *graph reduction* machines of various sorts, including the Id project at MIT (Arvind and Nikhil, 1987), the Rediflow project at Utah (Keller et al., 1979), the SK combinator machine SKIM at Cambridge (Stoye et al., 1984), the Manchester dataflow machine (Watson and Gurd, 1982), the ALICE parallel reduction machine at Imperial (Darlington and Reeve, 1981), the Burroughs NORMA combinator machine (Scheevel, 1986), and the DDM dataflow machine at Utah (Davis, 1977). Much (but not all) of this architecturally oriented work turned out to be a dead end, when it was later discovered that good compilers for stock architecture could outperform specialised architecture. But at the time it was all radical and exciting.

Several significant meetings took place in the early ’80s that lent additional impetus to the field.

In August 1980, the first Lisp conference took place in Stanford, California. Presentations included Rod Burstall, Dave MacQueen, and Don Sannella on Hope, the language that introduced algebraic data types (Burstall et al., 1980).

In July 1981, Peter Henderson, John Darlington, and David Turner ran an Advanced Course on Functional Programming and its Applications, in Newcastle (Darlington et al., 1982). All the big names were there: attendees included Gerry Sussman, Gary Lindstrom, David Park, Manfred Broy, Joe Stoy, and Edsger Dijkstra. (Hughes and Peyton Jones attended as students.) Dijkstra was characteristically unimpressed—he wrote “On the whole I could not avoid some feelings of deep disappointment. I still believe that the topic deserves a much more adequate treatment; quite a lot we were exposed to was definitely not up to par.” (Dijkstra, 1981)—but for many attendees it was a watershed.

In September 1981, the first conference on Functional Programming Languages and Computer Architecture (FPCA)—note the title!—took place in Portsmouth, New Hampshire. Here Turner gave his influential paper on “The semantic elegance of applicative languages” (Turner, 1981). (Wadler also presented his first conference paper.) FPCA became a key biennial conference in the field.

In September 1982, the second Lisp conference, now renamed Lisp and Functional Programming (LFP), took place in Pittsburgh,

Pennsylvania. Presentations included Peter Henderson on functional geometry (Henderson, 1982) and an invited talk by Turner on programming with infinite data structures. (It also saw the first published papers of Hudak, Hughes, and Peyton Jones.) Special guests at this conference included Church and Curry. The after-dinner talk was given by Barkley Rosser, and received two ovations in the middle, once when he presented the proof of Curry’s paradox, relating it to the Y combinator, and once when he presented a new proof of the Church-Rosser theorem. LFP became the other key biennial conference.

(In 1996, FPCA merged with LFP to become the annual International Conference on Functional Programming, ICFP, which remains the key conference in the field to the present day.)

In August 1987, Ham Richards of the University of Texas and David Turner organised an international school on Declarative Programming in Austin, Texas, as part of the UT “Year of Programming”. Speakers included: Samson Abramsky, John Backus, Richard Bird, Peter Buneman, Robert Cartwright, Simon Thompson, David Turner, and Hughes. A major part of the school was a course in lazy functional programming, with practical classes using Miranda.

All of this led to a tremendous sense of excitement. The simplicity and elegance of functional programming captivated the present authors, and many other researchers with them. Lazy evaluation—with its direct connection to the pure, call-by-name lambda calculus, the remarkable possibility of representing and manipulating infinite data structures, and addictively simple and beautiful implementation techniques—was like a drug.

(An anonymous reviewer supplied the following: “An interesting sidelight is that the Friedman and Wise paper inspired Sussman and Steele to examine lazy evaluation in Scheme, and for a time they weighed whether to make the revised version of Scheme call-by-name or call-by-value. They eventually chose to retain the original call-by-value design, reasoning that it seemed to be much easier to simulate call-by-name in a call-by-value language (using lambda-expressions as thunks) than to simulate call-by-value in a call-by-name language (which requires a separate evaluation-forcing mechanism). Whatever we might think of that reasoning, we can only speculate on how different the academic programming-language landscape might be today had they made the opposite decision.”)

2.2 A tower of Babel

As a result of all this activity, by the mid-1980s there were a number of researchers, including the authors, who were keenly interested in both design and implementation techniques for pure, lazy languages. In fact, many of us had independently designed our own lazy languages and were busily building our own implementations for them. We were each writing papers about our efforts, in which we first had to describe our languages before we could describe our implementation techniques. Languages that contributed to this lazy Tower of Babel include:

- Miranda, a successor to SASL and KRC, designed and implemented by David Turner using SK combinator reduction. While SASL and KRC were untyped, Miranda added strong polymorphic typing and type inference, ideas that had proven very successful in ML.
- Lazy ML (LML), pioneered at Chalmers by Augustsson and Johnsson, and taken up at University College London by Peyton Jones. This effort included the influential development of the *G-machine*, which showed that one could *compile* lazy functional programs to rather efficient code (Johnsson, 1984; Augustsson, 1984). (Although it is obvious in retrospect, we had become

used to the idea that laziness meant graph reduction, and graph reduction meant interpretation.)

- Orwell, a lazy language developed by Wadler, influenced by KRC and Miranda, and OL, a later variant of Orwell. Bird and Wadler co-authored an influential book on functional programming (Bird and Wadler, 1988), which avoided the “Tower of Babel” by using a more mathematical notation close to both Miranda and Orwell.
- Alfl, designed by Hudak, whose group at Yale developed a combinator-based interpreter for Alfl as well as a compiler based on techniques developed for Scheme and for T (a dialect of Scheme) (Hudak, 1984b; Hudak, 1984a).
- Id, a non-strict dataflow language developed at MIT by Arvind and Nikhil, whose target was a dataflow machine that they were building.
- Clean, a lazy language based explicitly on graph reduction, developed at Nijmegen by Rinus Plasmeijer and his colleagues (Brus et al., 1987).
- Ponder, a language designed by Jon Fairbairn, with an impredicative higher-rank type system and lexically scoped type variables that was used to write an operating system for SKIM (Fairbairn, 1985; Fairbairn, 1982).
- Daisy, a lazy dialect of Lisp, developed at Indiana by Cordelia Hall, John O’Donnell, and their colleagues (Hall and O’Donnell, 1985).

With the notable exception of Miranda (see Section 3.8), all of these were essentially single-site languages, and each individually lacked critical mass in terms of language-design effort, implementations, and users. Furthermore, although each had lots of interesting ideas, there were few reasons to claim that one language was demonstrably superior to any of the others. On the contrary, we felt that they were all roughly the same, bar the syntax, and we started to wonder why we didn’t have a single, common language that we could all benefit from.

At this time, both the Scheme and ML communities had developed their own standards. The Scheme community had major loci in MIT, Indiana, and Yale, and had just issued its ‘revised revised’ report (Rees and Clinger, 1986) (subsequent revisions would lead to the ‘revised⁵’, report (Kelsey et al., 1998)). Robin Milner had issued a ‘proposal for Standard ML’ (Milner, 1984) (which would later evolve into the definitive *Definition of Standard ML* (Milner and Tofte, 1990; Milner et al., 1997)), and Appel and MacQueen had released a new high-quality compiler for it (Appel and MacQueen, 1987).

2.3 The birth of Haskell

By 1987, the situation was akin to a supercooled solution—all that was needed was a random event to precipitate crystallisation. That event happened in the fall of ’87, when Peyton Jones stopped at Yale to see Hudak on his way to the 1987 Functional Programming and Computer Architecture Conference (FPCA) in Portland, Oregon. After discussing the situation, Peyton Jones and Hudak decided to initiate a meeting during FPCA, to garner interest in designing a new, common functional language. Wadler also stopped at Yale on the way to FPCA, and also endorsed the idea of a meeting.

The FPCA meeting thus marked the beginning of the Haskell design process, although we had no name for the language and very few technical discussions or design decisions occurred. In fact, a key point that came out of that meeting was that the easiest way to move forward was to begin with an existing language, and evolve

it in whatever direction suited us. Of all the lazy languages under development, David Turner's Miranda was by far the most mature. It was pure, well designed, fulfilled many of our goals, had a robust implementation as a product of Turner's company, Research Software Ltd, and was running at 120 sites. Turner was not present at the meeting, so we concluded that the first action item of the committee would be to ask Turner if he would allow us to adopt Miranda as the starting point for our new language.

After a brief and cordial interchange, Turner declined. His goals were different from ours. We wanted a language that could be used, among other purposes, for research into language features; in particular, we sought the freedom for anyone to extend or modify the language, and to build and distribute an implementation. Turner, by contrast, was strongly committed to maintaining a single language standard, with complete portability of programs within the Miranda community. He did not want there to be multiple dialects of Miranda in circulation and asked that we make our new language sufficiently distinct from Miranda that the two would not be confused. Turner also declined an invitation to join the new design committee.

For better or worse, this was an important fork in the road. Although it meant that we had to work through all the minutiae of a new language design, rather than starting from an already well-developed basis, it allowed us the freedom to contemplate more radical approaches to many aspects of the language design. For example, if we had started from Miranda it seems unlikely that we would have developed type classes (see Section 6.1). Nevertheless, Haskell owes a considerable debt to Miranda, both for general inspiration and specific language elements that we freely adopted where they fitted into our emerging design. We discuss the relationship between Haskell and Miranda further in Section 3.8.

Once we knew for sure that Turner would not allow us to use Miranda, an insanely active email discussion quickly ensued, using the mailing list `fplangc@cs.ucl.ac.uk`, hosted at the University College London, where Peyton Jones was a faculty member. The email list name came from the fact that originally we called ourselves the "FPLang Committee," since we had no name for the language. It wasn't until after we named the language (Section 2.4) that we started calling ourselves the "Haskell Committee."

2.4 The first meetings

The Yale Meeting The first physical meeting (after the impromptu FPCA meeting) was held at Yale, January 9–12, 1988, where Hudak was an Associate Professor. The first order of business was to establish the following goals for the language:

1. *It should be suitable for teaching, research, and applications, including building large systems.*
2. *It should be completely described via the publication of a formal syntax and semantics.*
3. *It should be freely available.* Anyone should be permitted to implement the language and distribute it to whomever they please.
4. *It should be usable as a basis for further language research.*
5. *It should be based on ideas that enjoy a wide consensus.*
6. *It should reduce unnecessary diversity in functional programming languages.* More specifically, we initially agreed to base it on an existing language, namely OL.

The last two goals reflected the fact that we intended the language to be quite conservative, rather than to break new ground. Although matters turned out rather differently, we intended to do little more

than embody the current consensus of ideas and to unite our disparate groups behind a single design.

As we shall see, not all of these goals were realised. We abandoned the idea of basing Haskell explicitly on OL very early; we violated the goal of embodying only well-tried ideas, notably by the inclusion of type classes; and we never developed a formal semantics. We discuss the way in which these changes took place in Section 3.

Directly from the minutes of the meeting, here is the committee process that we agreed upon:

1. Decide topics we want to discuss, and assign "lead person" to each topic.
2. Lead person begins discussion by summarising the issues for his topic.
 - In particular, begin with a description of how OL does it.
 - OL will be the default if no clearly better solution exists.
3. We should encourage breaks, side discussions, and literature research if necessary.
4. Some issues will *not* be resolved! But in such cases we should establish action items for their eventual resolution.
5. It may seem silly, but we should not adjourn this meeting until at least one thing is resolved: a *name* for the language!
6. Attitude will be important: a spirit of cooperation and compromise.

We return later to further discussion of the committee design process, in Section 3.5. A list of all people who served on the Haskell Committee appears in Section 14.

Choosing a Name The fifth item above was important, since a small but important moment in any language's evolution is the moment it is named. At the Yale meeting we used the following process (suggested by Wadler) for choosing the name.

Anyone could propose one or more names for the language, which were all written on a blackboard. At the end of this process, the following names appeared: Semla, Haskell, Vivaldi, Mozart, CFL (Common Functional Language), Funl 88, Semlor, Candle (Common Applicative Notation for Denoting Lambda Expressions), Fun, David, Nice, Light, ML Nouveau (or Miranda Nouveau, or LML Nouveau, or ...), Mirabelle, Concord, LL, Slim, Meet, Leval, Curry, Frege, Peano, Ease, Portland, and Haskell B Curry. After considerable discussion about the various names, each person was then free to cross out a name that he disliked. When we were done, there was one name left.

That name was "Curry," in honour of the mathematician and logician Haskell B. Curry, whose work had led, variously and indirectly, to our presence in that room. That night, two of us realised that we would be left with a lot of curry puns (aside from the spice, and the thought of currying favour, the one that truly horrified us was Tim Curry—TIM was Jon Fairbairn's abstract machine, and Tim Curry was famous for playing the lead in the Rocky Horror Picture Show). So the next day, after some further discussion, we settled on "Haskell" as the name for the new language. Only later did we realise that this was too easily confused with Pascal or Hasse!

Hudak and Wise were asked to write to Curry's widow, Virginia Curry, to ask if she would mind our naming the language after her husband. Hudak later visited Mrs. Curry at her home and listened to stories about people who had stayed there (such as Church and Kleene). Mrs. Curry came to his talk (which was about Haskell, of course) at Penn State, and although she didn't understand a word

of what he was saying, she was very gracious. Her parting remark was “You know, Haskell actually never liked the name Haskell.”

The Glasgow Meeting Email discussions continued fervently after the Yale Meeting, but it took a second meeting to resolve many of the open issues. That meeting was held April 6–9, 1988 at the University of Glasgow, whose functional programming group was beginning a period of rapid growth. It was at this meeting that many key decisions were made.

It was also agreed at this meeting that Hudak and Wadler would be the editors of the first Haskell Report. The name of the report, “Report on the Programming Language Haskell, A Non-strict, Purely Functional Language,” was inspired in part by the “Report on the Algorithmic Language Scheme,” which in turn was modelled after the “Report on the Algorithmic Language Algol.”

IFIP WG2.8 Meetings The ’80s were an exciting time to be doing functional programming research. One indication of that excitement was the establishment, due largely to the effort of John Williams (long-time collaborator with John Backus at IBM Almaden), of IFIP Working Group 2.8 on Functional Programming. This not only helped to bring legitimacy to the field, it also provided a convenient venue for talking about Haskell and for piggy-backing Haskell Committee meetings before or after WG2.8 meetings. The first two WG2.8 meetings were held in Glasgow, Scotland, July 11–15, 1988, and in Mystic, CT, USA, May 1–5, 1989 (Mystic is about 30 minutes from Yale). Figure 1 was taken at the 1992 meeting of WG2.8 in Oxford.

2.5 Refining the design

After the initial flurry of face-to-face meetings, there followed fifteen years of detailed language design and development, coordinated entirely by electronic mail. Here is a brief time-line of how Haskell developed:

September 1987. Initial meeting at FPCA, Portland, Oregon.

December 1987. Subgroup meeting at University College London.

January 1988. A multi-day meeting at Yale University.

April 1988. A multi-day meeting at the University of Glasgow.

July 1988. The first IFIP WG2.8 meeting, in Glasgow.

May 1989. The second IFIP WG2.8 meeting, in Mystic, CT.

1 April 1990. The Haskell version 1.0 Report was published (125 pages), edited by Hudak and Wadler. At the same time, the Haskell mailing list was started, open to all.

The closed `fplangc` mailing list continued for committee discussions, but increasingly debate took place on the public Haskell mailing list. Members of the committee became increasingly uncomfortable with the “us-and-them” overtones of having both public and private mailing lists, and by April 1991 the `fplangc` list fell into disuse. All further discussion about Haskell took place in public, but decisions were still made by the committee.

August 1991. The Haskell version 1.1 Report was published (153 pages), edited by Hudak, Peyton Jones, and Wadler. This was mainly a “tidy-up” release, but it included `let` expressions and operator sections for the first time.

March 1992. The Haskell version 1.2 Report was published (164 pages), edited by Hudak, Peyton Jones, and Wadler, introducing only minor changes to Haskell 1.1. Two months later, in May 1992, it appeared in *SIGPLAN Notices*, accompanied by a “Gentle introduction to Haskell” written by Hudak and Fasel. We are very grateful to the SIGPLAN chair Stu Feldman, and

the *Notices* editor Dick Wexelblat, for their willingness to publish such an enormous document. It gave Haskell both visibility and credibility.

1994. Haskell gained Internet presence when John Peterson registered the `haskell.org` domain name and set up a server and website at Yale. (Hudak’s group at Yale continues to maintain the `haskell.org` server to this day.)

May 1996. The Haskell version 1.3 Report was published, edited by Hammond and Peterson. In terms of technical changes, Haskell 1.3 was the most significant release of Haskell after 1.0. In particular:

- A Library Report was added, reflecting the fact that programs can hardly be portable unless they can rely on standard libraries.
- Monadic I/O made its first appearance, including “do” syntax (Section 7), and the I/O semantics in the Appendix was dropped.
- Type classes were generalised to higher kinds—so-called “constructor classes” (see Section 6).
- Algebraic data types were extended in several ways: newtypes, strictness annotations, and named fields.

April 1997. The Haskell version 1.4 report was published (139 + 73 pages), edited by Peterson and Hammond. This was a tidy-up of the 1.3 report; the only significant change is that list comprehensions were generalised to arbitrary monads, a decision that was reversed two years later.

February 1999 The Haskell 98 Report: Language and Libraries was published (150 + 89 pages), edited by Peyton Jones and Hughes. As we describe in Section 3.7, this was a very significant moment because it represented a commitment to stability. List comprehensions reverted to just lists.

1999–2002 In 1999 the Haskell Committee *per se* ceased to exist. Peyton Jones took on sole editorship, with the intention of collecting and fixing typographical errors. Decisions were no longer limited to a small committee; now anyone reading the Haskell mailing list could participate.

However, as Haskell became more widely used (partly because of the existence of the Haskell 98 standard), many small flaws emerged in the language design, and many ambiguities in the Report were discovered. Peyton Jones’s role evolved to that of Benign Dictator of Linguistic Minutiae.

December 2002 The Revised Haskell 98 Report: Language and Libraries was published (260 pages), edited by Peyton Jones. Cambridge University Press generously published the Report as a book, while agreeing that the entire text could still be available online and be freely usable in that form by anyone. Their flexibility in agreeing to publish a book under such unusual terms was extraordinarily helpful to the Haskell community, and defused a tricky debate about freedom and intellectual property.

It is remarkable that it took four years from the first publication of Haskell 98 to “shake down” the specification, even though Haskell was already at least eight years old when Haskell 98 came out. Language design is a slow process!

Figure 2 gives the Haskell time-line in graphical form¹. Many of the implementations, libraries, and tools mentioned in the figure are discussed later in the paper.

¹This figure was kindly prepared by Bernie Pope and Don Stewart.



Back row	John Launchbury, Neil Jones, Sebastian Hunt, Joe Fasel, Geraint Jones (glasses), Geoffrey Burn, Colin Runciman (moustache)
Next row	Philip Wadler (big beard), Jack Dennis (beard), Patrick O'Keefe (glasses), Alex Aiken (mostly hidden), Richard Bird, Lennart Augustsson, Rex Page, Chris Hankin (moustache), Joe Stoy (red shirt), John Williams, John O'Donnell, David Turner (red tie)
Front standing row	Mario Coppo, Warren Burton, Corrado Boehm, Dave MacQueen (beard), Mary Sheeran, John Hughes, David Lester
Seated	Karen MacQueen, Luca Cardelli, Dick Kieburz, Chris Clack, Mrs Boehm, Mrs Williams, Dorothy Peyton Jones
On floor	Simon Peyton Jones, Paul Hudak, Richard (Corky) Cartwright

Figure 1. Members and guests of IFIP Working Group 2.8, Oxford, 1992

2.6 Was Haskell a joke?

The first edition of the Haskell Report was published on April 1, 1990. It was mostly an accident that it appeared on April Fool's Day—a date had to be chosen, and the release was close enough to April 1 to justify using that date. Of course Haskell was no joke, but the release did lead to a number of subsequent April Fool's jokes.

What got it all started was a rather frantic year of Haskell development in which Hudak's role as editor of the Report was especially stressful. On April 1 a year or two later, he sent an email message to the Haskell Committee saying that it was all too much for him, and that he was not only resigning from the committee, he was also quitting Yale to pursue a career in music. Many members of the committee bought into the story, and David Wise immediately phoned Hudak to plead with him to reconsider his decision.

Of course it was just an April Fool's joke, but the seed had been planted for many more to follow. Most of them are detailed on the Haskell website at haskell.org/humor, and here is a summary of the more interesting ones:

1. On April 1, 1993, Will Partain wrote a brilliant announcement about an extension to Haskell called *Haskerl* that combined the best ideas in Haskell with the best ideas in Perl. Its technically detailed and very serious tone made it highly believable.

2. Several of the responses to Partain's well-written hoax were equally funny, and also released on April 1. One was by Hudak, in which he wrote:

“Recently Haskell was used in an experiment here at Yale in the Medical School. It was used to replace a C program that controlled a heart-lung machine. In the six months that it was in operation, the hospital estimates that probably a dozen lives were saved because the program was far more robust than the C program, which often crashed and killed the patients.”

In response to this, Nikhil wrote:

“Recently, a local hospital suffered many malpractice suits due to faulty software in their X-ray machine. So, they decided to rewrite the code in Haskell for more reliability.”

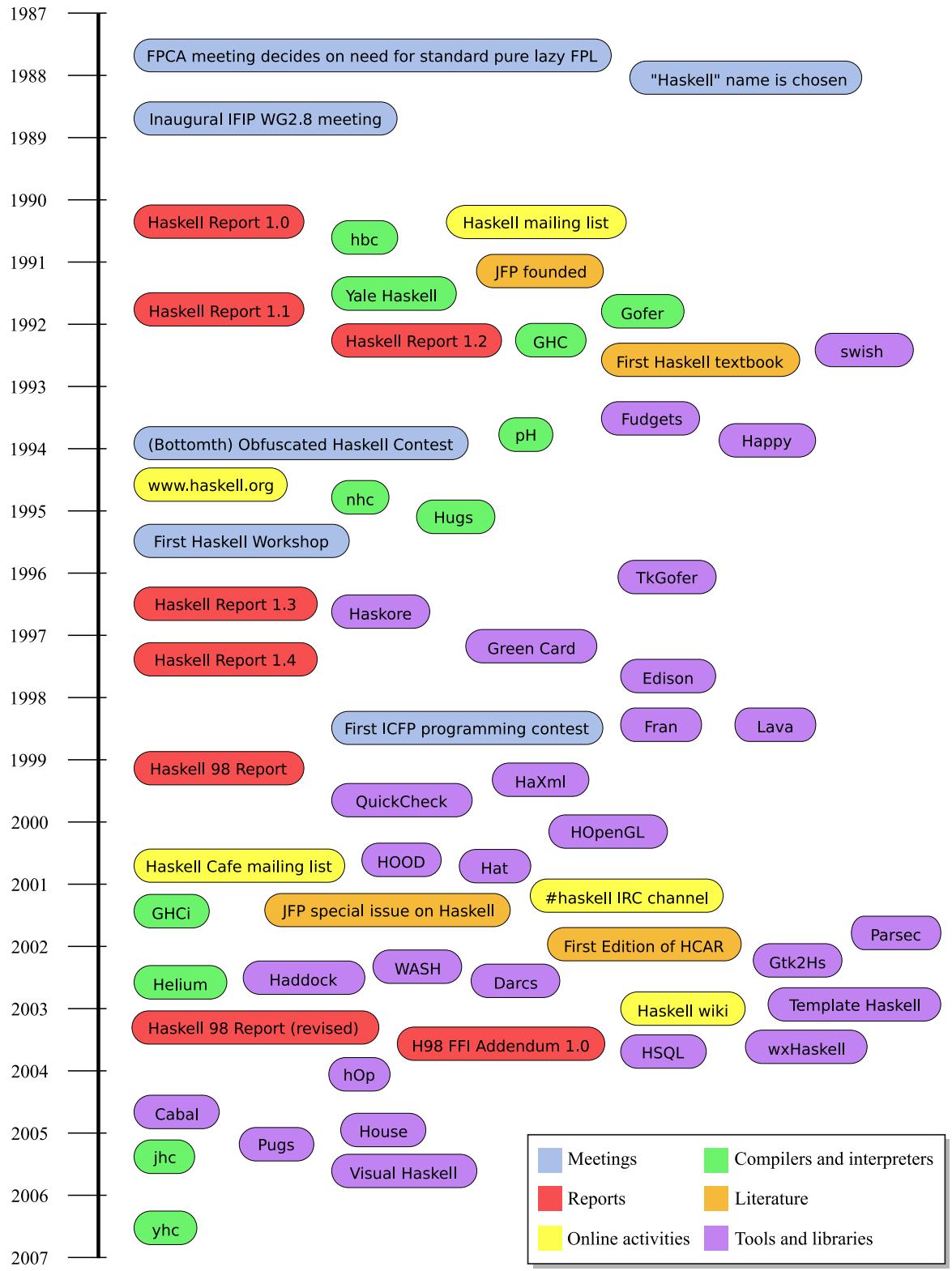


Figure 2. Haskell timeline

“Malpractice suits have now dropped to zero. The reason is that they haven’t taken any new X-rays (‘we’re still compiling the Standard Prelude’).”

3. On April 1, 1998, John Peterson wrote a bogus press release in which it was announced that because Sun Microsystems had sued Microsoft over the use of Java, Microsoft had decided to adopt Haskell as its primary software development language. Ironically, not long after this press release, Peyton Jones announced his move from Glasgow to Microsoft Research in Cambridge, an event that Peterson knew nothing about at the time.

Subsequent events have made Peterson’s jape even more prophetic. Microsoft did indeed respond to Java by backing another language, but it was C# rather than Haskell. But many of the features in C# were pioneered by Haskell and other functional languages, notably polymorphic types and LINQ (Language Integrated Query). Erik Meijer, a principal designer of LINQ, says that LINQ is directly inspired by the monad comprehensions in Haskell.

4. On April 1, 2002, Peterson wrote another bogus but entertaining and plausible article entitled “Computer Scientist Gets to the ‘Bottom’ of Financial Scandal.” The article describes how Peyton Jones, using his research on formally valuating financial contracts using Haskell (Peyton Jones et al., 2000), was able to unravel Enron’s seedy and shaky financial network. Peyton Jones is quoted as saying:

“It’s really very simple. If I write a contract that says its value is derived from a stock price and the worth of the stock depends solely on the contract, we have bottom. So in the end, Enron had created a complicated series of contracts that ultimately had no value at all.”

3. Goals, principles, and processes

In this section we reflect on the principles that underlay our thinking, the big choices that we made, and processes that led to them.

3.1 Haskell is lazy

Laziness was undoubtedly the single theme that united the various groups that contributed to Haskell’s design. Technically, Haskell is a language with a non-strict semantics; lazy evaluation is simply one implementation technique for a non-strict language. Nevertheless the term “laziness” is more pungent and evocative than “non-strict,” so we follow popular usage by describing Haskell as lazy. When referring specifically to implementation techniques we will use the term “call-by-need,” in contrast with the call-by-value mechanism of languages like Lisp and ML.

By the mid-eighties, there was almost a decade of experience of lazy functional programming in practice, and its attractions were becoming better understood. Hughes’s paper “Why functional programming matters” captured these in an influential manifesto for lazy programming, and coincided with the early stages of Haskell’s design. (Hughes first presented it as his interview talk when applying for a position at Oxford in 1984, and it circulated informally before finally being published in 1989 (Hughes, 1989).)

Laziness has its costs. Call-by-need is usually less efficient than call-by-value, because of the extra bookkeeping required to delay evaluation until a term is required, so that some terms may not be evaluated, and to overwrite a term with its value, so that no term is evaluated twice. This cost is a significant but constant factor, and was understood at the time Haskell was designed.

A much more important problem is this: it is very hard for even experienced programmers to predict the *space* behaviour of lazy

programs, and there can be much more than a constant factor at stake. As we discuss in Section 10.2, the prevalence of these space leaks led us to add some strict features to Haskell, such as `seq` and strict data types (as had been done in SASL and Miranda earlier). Dually, strict languages have dabbled with laziness (Wadler et al., 1988). As a result, the strict/lazy divide has become much less an all-or-nothing decision, and the practitioners of each recognise the value of the other.

3.2 Haskell is pure

An immediate consequence of laziness is that evaluation order is demand-driven. As a result, it becomes more or less impossible to reliably perform input/output or other side effects as the result of a function call. Haskell is, therefore, a *pure* language. For example, if a function `f` has type `Int → Int` you can be sure that `f` will not read or write any mutable variables, nor will it perform any input/output. In short, `f` really is a *function* in the mathematical sense: every call (`f 3`) will return the same value.

Once we were committed to a *lazy* language, a *pure* one was inescapable. The converse is not true, but it is notable that in practice most pure programming languages are also lazy. Why? Because in a call-by-value language, whether functional or not, the temptation to allow unrestricted side effects inside a “function” is almost irresistible.

Purity is a big bet, with pervasive consequences. Unrestricted side effects are undoubtedly very convenient. Lacking side effects, Haskell’s input/output was initially painfully clumsy, which was a source of considerable embarrassment. Necessity being the mother of invention, this embarrassment ultimately led to the invention of *monadic I/O*, which we now regard as one of Haskell’s main contributions to the world, as we discuss in more detail in Section 7.

Whether a pure language (with monadic effects) is ultimately the best way to write programs is still an open question, but it certainly is a radical and elegant attack on the challenge of programming, and it was that combination of power and beauty that motivated the designers. In retrospect, therefore, perhaps the biggest single benefit of laziness is not laziness *per se*, but rather that laziness kept us pure, and thereby motivated a great deal of productive work on monads and encapsulated state.

3.3 Haskell has type classes

Although laziness was what brought Haskell’s designers together, it is perhaps type classes that are now regarded as Haskell’s most distinctive characteristic. Type classes were introduced to the Haskell Committee by Wadler in a message sent to the `fplangc` mailing list dated 24 February 1988.

Initially, type classes were motivated by the narrow problem of overloading of numeric operators and equality. These problems had been solved in completely different ways in Miranda and SML.

SML used overloading for the built-in numeric operators, resolved at the point of call. This made it hard to define new numeric operations in terms of old. If one wanted to define, say, square in terms of multiplication, then one had to define a different version for each numeric type, say integers and floats. Miranda avoided this problem by having only a single numeric type, called `num`, which was a union of unbounded-size integers and double-precision floats, with automatic conversion of `int` to `float` when required. This is convenient and flexible but sacrifices some of the advantages of static typing – for example, in Miranda the expression `(mod 8 3.4)` is type-correct, even though in most languages the modulus operator `mod` only makes sense for integer moduli.

SML also originally used overloading for equality, so one could not define the polymorphic function that took a list and a value and returned true if the value was equal to some element of the list. (To define this function, one would have to pass in an equality-testing function as an extra argument.) Miranda simply gave equality a polymorphic type, but this made equality well defined on function types (it raised an error at run time) and on abstract types (it compared their underlying representation for equality, a violation of the abstraction barrier). A later version of SML included polymorphic equality, but introduced special “equality type variables” (written ‘ λ instead of ‘ a) that ranged only over types for which equality was defined (that is, not function types or abstract types).

Type classes provided a uniform solution to both of these problems. They generalised the notion of equality type variables from SML, introducing a notion of a “class” of types that possessed a given set of operations (such as numeric operations or equality).

The type-class solution was attractive to us because it seemed more principled, systematic and modular than any of the alternatives; so, despite its rather radical and unproven nature, it was adopted by acclamation. Little did we know what we were letting ourselves in for!

Wadler conceived of type classes in a conversation with Joe Fasel after one of the Haskell meetings. Fasel had in mind a different idea, but it was he who had the key insight that overloading should be reflected in the type of the function. Wadler misunderstood what Fasel had in mind, and type classes were born! Wadler’s student Steven Blott helped to formulate the type rules, and proved the system sound, complete, and coherent for his doctoral dissertation (Wadler and Blott, 1989; Blott, 1991). A similar idea was formulated independently by Stefan Kaes (Kaes, 1988).

We elaborate on some of the details and consequences of the type-class approach in Section 6. Meanwhile, it is instructive to reflect on the somewhat accidental nature of such a fundamental and far-reaching aspect of the Haskell language. It was a happy coincidence of timing that Wadler and Blott happened to produce this key idea at just the moment when the language design was still in flux. It was adopted, with little debate, in direct contradiction to our implicit goal of embodying a tried-and-tested consensus. It had far-reaching consequences that dramatically exceeded our initial reason for adopting it in the first place.

3.4 Haskell has no formal semantics

One of our explicit goals was to produce a language that had a formally defined type system and semantics. We were strongly motivated by mathematical techniques in programming language design. We were inspired by our brothers and sisters in the ML community, who had shown that it was possible to give a complete formal definition of a language, and the *Definition of Standard ML* (Milner and Tofte, 1990; Milner et al., 1997) had a place of honour on our shelves.

Nevertheless, we never achieved this goal. The Haskell Report follows the usual tradition of language definitions: it uses carefully worded English language. Parts of the language (such as the semantics of pattern matching) are defined by a translation into a small “core language”, but the latter is never itself formally specified. Subsequent papers describe a good part of Haskell, especially its type system (Faxen, 2002), but there is no one document that describes the whole thing. Why not? Certainly not because of a conscious choice by the Haskell Committee. Rather, it just never seemed to be the most urgent task. No one undertook the work, and in practice the language users and implementers seemed to manage perfectly well without it.

Indeed, in practice the static semantics of Haskell (i.e. the semantics of its type system) is where most of the complexity lies. The consequences of not having a formal static semantics is perhaps a challenge for compiler writers, and sometimes results in small differences between different compilers. But for the user, once a program type-checks, there is little concern about the static semantics, and little need to reason formally about it.

Fortunately, the dynamic semantics of Haskell is relatively simple. Indeed, at many times during the design of Haskell, we resorted to denotational semantics to discuss design options, as if we all knew what the semantics of Haskell *should* be, even if we didn’t write it all down formally. Such reasoning was especially useful in reasoning about “bottom” (which denotes error or non-termination and occurs frequently in a lazy language in pattern matching, function calls, recursively defined values, and so on).

Perhaps more importantly, the dynamic semantics of Haskell is captured very elegantly for the average programmer through “equational reasoning”—much simpler to apply than a formal denotational or operational semantics, thanks to Haskell’s purity. The theoretical basis for equational reasoning derives from the standard reduction rules in the lambda calculus (β - and η -reduction), along with those for primitive operations (so-called δ -rules). Combined with appropriate induction (and co-induction) principles, it is a powerful reasoning method in practice. Equational reasoning in Haskell is part of the culture, and part of the training that every good Haskell programmer receives. As a result, there may be more proofs of correctness properties and program transformations in Haskell than any other language, despite its lack of a formally specified semantics! Such proofs usually ignore the fact that some of the basic steps used—such as η -reduction in Haskell—would not actually preserve a fully formal semantics even if there was one, yet amazingly enough, (under the right conditions) the conclusions drawn are valid even so (Danielsson et al., 2006)!

Nevertheless, we always found it a little hard to admit that a language as principled as Haskell aspires to be has no formal definition. But that is the fact of the matter, and it is not without its advantages. In particular, the absence of a formal language definition does allow the language to *evolve* more easily, because the costs of producing fully formal specifications of any proposed change are heavy, and by themselves discourage changes.

3.5 Haskell is a committee language

Haskell is a language designed by committee, and conventional wisdom would say that a committee language will be full of warts and awkward compromises. In a memorable letter to the Haskell Committee, Tony Hoare wistfully remarked that Haskell was “probably doomed to succeed.”

Yet, as it turns out, for all its shortcomings Haskell is often described as “beautiful” or “elegant”—even “cool”—which are hardly words one would usually associate with committee designs. How did this come about? In reflecting on this question we identified several factors that contributed:

- The initial situation, described above in Section 2, was very favourable. Our individual goals were well aligned, and we began with a strong shared, if somewhat fuzzy, vision of what we were trying to achieve. We all needed Haskell.
- Mathematical elegance was extremely important to us, formal semantics or no formal semantics. Many debates were punctuated by cries of “does it have a compositional semantics?” or “what does the domain look like?” This semi-formal approach certainly made it more difficult for *ad hoc* language features to creep in.

- We held several multi-day face-to-face meetings. Many matters that were discussed extensively by email were only resolved at one of these meetings.
- At each moment in the design process, one or two members of the committee served as *The Editor*. The Editor could not make binding decisions, but was responsible for driving debates to a conclusion. He also was the custodian of the Report, and was responsible for embodying the group’s conclusion in it.
- At each moment in the design process, one member of the committee (not necessarily the Editor) served as the *Syntax Czar*. The Czar was empowered to make binding decisions about syntactic matters (only). Everyone always says that far too much time is devoted to discussing syntax—but many of the same people will fight to the death for their preferred symbol for lambda. The Syntax Czar was our mechanism for bringing such debates to an end.

3.6 Haskell is a big language

A major source of tension both within and between members of the committee was the competition between beauty and utility. On the one hand we passionately wanted to design a simple, elegant language; as Hoare so memorably put it, “There are two ways of constructing a software design: one way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.” On the other hand, we also *really* wanted Haskell to be a useful language, for both teaching and real applications.

Although very real, this dilemma never led to open warfare. It did, however, lead Richard Bird to resign from the committee in mid-1988, much to our loss. At the time he wrote, “On the evidence of much of the material and comments submitted to fplang, there is a severe danger that the principles of simplicity, ease of proof, and elegance will be overthrown. Because much of what is proposed is half-baked, retrogressive, and even baroque, the result is likely to be a mess. We are urged to return to the mind-numbing syntax of Lisp (a language that held back the pursuit of functional programming for over a decade). We are urged to design for ‘big’ programs, because constructs that are ‘aesthetic’ for small programs will lose their attractiveness when the scale is increased. We are urged to allow large where-clauses with deeply nested structures. In short, it seems we are urged to throw away the one feature of functional programming that distinguishes it from the conventional kind and may ensure its survival into the 21st century: susceptibility to formal proof and construction.”

In the end, the committee wholeheartedly embraced *superficial* complexity; for example, the syntax supports many ways of expressing the same thing, in contradiction to our original inclinations (Section 4.4). In other places, we eschewed *deep* complexity, despite the cost in expressiveness—for example, we avoided parametrised modules (Section 8.2) and extensible records (Section 5.6). In just one case, type classes, we adopted an idea that complicated everything but was just too good to miss. The reader will have to judge the resulting balance, but even in retrospect we feel that the elegant core of purely functional programming has survived remarkably unscathed. If we had to pick places where real compromises were made, they would be the monomorphism restriction (see Section 6.2) and the loss of parametricity, currying, and surjective pairing due to `seq` (see Section 10.3).

3.7 Haskell and Haskell 98

The goal of using Haskell for research demands *evolution*, while using the language for teaching and applications requires *stability*. At the beginning, the emphasis was firmly on evolution. The preface of every version of the Haskell Report states: “*The committee hopes that Haskell can serve as a basis for future research in language design. We hope that extensions or variants of the language may appear, incorporating experimental features.*”

However, as Haskell started to become popular, we started to get complaints about changes in the language, and questions about what our plans were. “I want to write a book about Haskell, but I can’t do that if the language keeps changing” is a typical, and fully justified, example.

In response to this pressure, the committee evolved a simple and obvious solution: we simply named a particular instance of the language “Haskell 98,” and language implementers committed themselves to continuing to support Haskell 98 indefinitely. We regarded Haskell 98 as a reasonably conservative design. For example, by that time multi-parameter type classes were being widely used, but Haskell 98 only has single-parameter type classes (Peyton Jones et al., 1997).

The (informal) standardisation of Haskell 98 was an important turning point for another reason: it was the moment that the Haskell Committee disbanded. There was (and continues to be) a tremendous amount of innovation and activity in the Haskell community, including numerous proposals for language features. But rather than having a committee to choose and bless particular ones, it seemed to us that the best thing to do was to get out of the way, let a thousand flowers bloom, and see which ones survived. It was also a huge relief to be able to call the task finished and to file our enormous mail archives safely away.

We made no attempt to discourage variants of Haskell other than Haskell 98; on the contrary, we explicitly encouraged the further development of the language. The nomenclature encourages the idea that “Haskell 98” is a stable variant of the language, while its free-spirited children are free to term themselves “Haskell.”

In the absence of a language committee, Haskell has continued to evolve apace, in two quite different ways.

- First, as Haskell has become a mature language with thousands of users, it has had to grapple with the challenges of scale and complexity with which any real-world language is faced. That has led to a range of practically oriented features and resources, such as a foreign-function interface, a rich collection of libraries, concurrency, exceptions, and much else besides. We summarise these developments in Section 8.
- At the same time, the language has simultaneously served as a highly effective laboratory in which to explore advanced language design ideas, especially in the area of type systems and meta-programming. These ideas surface both in papers—witness the number of research papers that take Haskell as their base language—and in Haskell implementations. We discuss a number of examples in Section 6.

The fact that Haskell has, thus far, managed the tension between these two strands of development is perhaps due to an accidental virtue: Haskell has not become *too* successful. The trouble with runaway success, such as that of Java, is that you get too many users, and the language becomes bogged down in standards, user groups, and legacy issues. In contrast, the Haskell community is small enough, and agile enough, that it usually not only absorbs language changes but positively welcomes them: it’s like throwing red meat to hyenas.

3.8 Haskell and Miranda

At the time Haskell was born, by far the most mature and widely used non-strict functional language was Miranda. Miranda was a product of David Turner's company, Research Software Limited, which he founded in 1983. Turner conceived Miranda to carry lazy functional programming, with Hindley-Milner typing (Milner, 1978), into the commercial domain. First released in 1985, with subsequent releases in 1987 and 1989, Miranda had a well supported implementation, a nice interactive user interface, and a variety of textbooks (four altogether, of which the first was particularly influential (Bird and Wadler, 1988)). It was rapidly taken up by both academic and commercial licences, and by the early 1990s Miranda was installed (although not necessarily taught) at 250 universities and around 50 companies in 20 countries.

Haskell's design was, therefore, strongly influenced by Miranda. At the time, Miranda was the fullest expression of a non-strict, purely functional language with a Hindley-Milner type system and algebraic data types—and that was precisely the kind of language that Haskell aspired to be. As a result, there are many similarities between the two languages, both in their basic approach (purity, higher order, laziness, static typing) and in their syntactic look and feel. Examples of the latter include: the equational style of function definitions, especially pattern matching, guards, and `where` clauses; algebraic types; the notation for lists and list comprehensions; writing pair types as `(num, bool)` rather than the `int*bool` of ML; capitalisation of data constructors; lexically distinguished user-defined infix operators; the use of a layout rule; and the naming of many standard functions.

There are notable differences from Miranda too, including: placement of guards on the left of “`=`” in a definition; a richer syntax for expressions (Section 4.4); different syntax for data type declarations; capitalisation of type constructors as well as data constructors; use of alphanumeric identifiers for type variables, rather than Miranda's `*`, `**`, etc.; how user-defined operators are distinguished (`x $op y` in Miranda vs. `x `op` y` in Haskell); and the details of the layout rule. More fundamentally, Haskell did not adopt Miranda's abstract data types, using the module system instead (Section 5.3); added monadic I/O (Section 7.2); and incorporated many innovations to the core Hindley-Milner type system, especially type classes (Section 6).

Today, Miranda has largely been displaced by Haskell. One indication of that is the publication of textbooks: while Haskell books continue to appear regularly, the last textbook in English to use Miranda was published in 1995. This is at first sight surprising, because it can be hard to displace a well-established incumbent, but the economics worked against Miranda: Research Software was a small company seeking a return on its capital; academic licences were cheaper than commercial ones, but neither were free, while Haskell was produced by a group of universities with public funds and available free to academic and commercial users alike. Moreover, Miranda ran only under Unix, and the absence of a Windows version increasingly worked against it.

Although Miranda initially had the better implementation, Haskell implementations improved more rapidly—it was hard for a small company to keep up. Hugs gave Haskell a fast interactive interface similar to that which Research Software supplied for Miranda (and Hugs ran under both Unix and Windows), while Moore's law made Haskell's slow compilers acceptably fast and the code they generated even faster. And Haskell had important new ideas, as this paper describes. By the mid-1990s, Haskell was a much more practical choice for real programming than Miranda.

Miranda's proprietary status did not enjoy universal support in the academic community. As required to safeguard his trademark, Turner always footnoted the first occurrence of Miranda in his papers to state it was a trademark of Research Software Limited. In response, some early Haskell presentations included a footnote “Haskell is not a trademark”. Miranda's licence conditions at that time required the licence holder to seek permission before distributing an implementation of Miranda or a language whose design was substantially copied from Miranda. This led to friction between Oxford University and Research Software over the possible distribution of Wadler's language Orwell. However, despite Haskell's clear debt to Miranda, Turner raised no objections to Haskell.

The tale raises a tantalising “what if” question. What if David Turner had placed Miranda in the public domain, as some urged him to do? Would the mid '80s have seen a standard lazy functional language, supported by the research community *and* with a company backing it up? Could Research Software have found a business model that enabled it to benefit, rather than suffer, from university-based implementation efforts? Would the additional constraints of an existing design have precluded the creative and sometimes anarchic ferment that has characterised the Haskell community? How different could history have been?

Miranda was certainly no failure, either commercially or scientifically. It contributed a small, elegant language design with a well-supported implementation, which was adopted in many universities and undoubtedly helped encourage the spread of functional programming in university curricula. Beyond academia, the use of Miranda in several large projects (Major and Turcotte, 1991; Page and Moe, 1993) demonstrated the industrial potential of a lazy functional language. Miranda is still in use today: it is still taught in some institutions, and the implementations for Linux and Solaris (now free) continue to be downloaded. Turner's efforts made a permanent and valuable contribution to the development of interest in the subject in general, paving the way for Haskell a few years later.

Part II

Technical Contributions

4. Syntax

The phrase “syntax is not important” is often heard in discussions about programming languages. In fact, in the 1980s this phrase was heard more often than it is today, partly because there was so much interest at the time in developing the theory behind, and emphasising the importance of, the *formal semantics* of programming languages, which was a relatively new field in itself. Many programming language researchers considered syntax to be the trivial part of language design, and semantics to be “where the action was.”

Despite this, the Haskell Committee worked very hard—meaning it spent endless hours—on designing (and arguing about) the syntax of Haskell. It wasn't so much that we were boldly bucking the trend, or that the phrase “syntax is important” was a new retro-phrase that became part of our discourse, but rather that, for better or worse, we found that syntax design could be not only fun, but an obsession. We also found that syntax, being the user interface of a language, could become very personal. There is no doubt that some of our most heated debates were over syntax, not semantics.

In the end, was it worth it? Although not an explicit goal, one of the most pleasing consequences of our effort has been comments heard

many times over the years that “Haskell is a pretty language.” For some reason, many people think that Haskell programs look nice. Why is that? In this section we give historical perspectives on many of the syntactic language features that we think contribute to this impression. Further historical details, including some ideas considered and ultimately rejected, may be found in Hudak’s *Computing Surveys* article (Hudak, 1989).

4.1 Layout

Most imperative languages use a semicolon to separate sequential commands. In a language without side effects, however, the notion of sequencing is completely absent. There is still the need to separate declarations of various kinds, but the feeling of the Haskell Committee was that we should avoid the semicolon and its sequential, imperative baggage.

Exploiting the physical layout of the program text is a simple and elegant way to avoid syntactic clutter. We were familiar with the idea, in the form of the “offside rule” from our use of Turner’s languages SASL (Turner, 1976) and Miranda (Turner, 1986), although the idea goes back to Christopher Strachey’s CPL (Barron et al., 1963), and it was also featured in ISWIM (Landin, 1966).

The layout rules needed to be very simple, otherwise users would object, and we explored many variations. We ended up with a design that differed from our most immediate inspiration, Miranda, in supporting larger function definitions with less enforced indentation. Although we felt that good programming style involved writing small, short function definitions, in practice we expected that programmers would also want to write fairly large function definitions—and it would be a shame if layout got in the way. So Haskell’s layout rules are considerably more lenient than Miranda’s in this respect. Like Miranda, we provided a way for the user to override implicit layout selectively, in our case by using explicit curly braces and semicolons instead. One reason we thought this was important is that we expected people to write programs that generated Haskell programs, and we thought it would be easier to generate explicit separators than layout.

Influenced by these constraints and a desire to “do what the programmer expects”, Haskell evolved a fairly complex layout rule—complex enough that it was formally specified for the first time in the Haskell 98 Report. However, after a short adjustment period, most users find it easy to adopt a programming style that falls within the layout rules, and rarely resort to overriding them².

4.2 Functions and function application

There are lots of ways to define functions in Haskell—after all, it is a functional language—but the ways are simple and all fit together in a sensible manner.

Currying Following a tradition going back to Frege, a function of two arguments may be represented as a function of one argument that itself returns a function of one argument. This tradition was honed by Moses Schönfinkel and Haskell Curry and came to be called *currying*.

Function application is denoted by juxtaposition and associates to the left. Thus, $f\ x\ y$ is parsed $(f\ x)\ y$. This leads to concise and powerful code. For example, to square each number in a list we write `map square [1,2,3]`, while to square each number in a list of lists we write `map (map square) [[1,2],[3]]`.

Haskell, like many other languages based on lambda calculus, supports both curried and uncurried definitions:

```
hyp :: Float -> Float -> Float
hyp x y = sqrt (x*x + y*y)
```

```
hyp :: (Float, Float) -> Float
hyp (x,y) = sqrt (x*x + y*y)
```

In the latter, the function is viewed as taking a single argument, which is a pair of numbers. One advantage of currying is that it is often more compact: $f\ x\ y$ contains three fewer lexemes than $f(x,y)$.

Anonymous functions The syntax for anonymous functions, $\lambda x \rightarrow exp$, was chosen to resemble lambda expressions, since the backslash was the closest single ASCII character to the Greek letter λ . However, “ \rightarrow ” was used instead of a period in order to reserve the period for function composition.

Prefix operators Haskell has only one prefix operator: arithmetic negation. The Haskell Committee in fact did not want *any* prefix operators, but we couldn’t bring ourselves to force users to write something like `minus 42` or `~42` for the more conventional `-42`. Nevertheless, the dearth of prefix operators makes it easier for readers to parse expressions.

Infix operators The Haskell Committee wanted expressions to look as much like mathematics as possible, and thus from day one we bought into the idea that Haskell would have infix operators.³ It was also important to us that infix operators be definable by the user, including declarations of precedence and associativity. Achieving all this was fairly conventional, but we also defined the following simple relationship between infix application and conventional function application: the former *always* binds less tightly than the latter. Thus $f\ x + g\ y$ never needs parentheses, regardless of what infix operator is used. This design decision proved to be a good one, as it contributes to the readability of programs. (Sadly, this simple rule is not adhered to by @-patterns, which bind more tightly than anything; this was probably a mistake, although @-patterns are not used extensively enough to cause major problems.)

Sections Although a commitment to infix operators was made quite early, there was also the feeling that all values in Haskell should be “first class”—especially functions. So there was considerable concern about the fact that infix operators were not, by themselves, first class, a problem made apparent by considering the expression $f + x$. Does this mean the function f applied to two arguments, or the function $+$ applied to two arguments?

The solution to this problem was to use a generalised notion of *sections*, a notation that first appeared in David Wile’s dissertation (Wile, 1973) and was then disseminated via IFIP WG2.1—among others to Bird, who adopted it in his work, and Turner, who introduced it into Miranda. A section is a partial application of an infix operator to no arguments, the left argument, or the right argument—and by surrounding the result in parentheses, one then has a first-class functional value. For example, the following equivalences hold:

```
(+) = \x y -> x+y
(x+) = \y -> x+y
(+y) = \x -> x+y
```

Being able to partially apply infix operators is consistent with being able to partially apply curried functions, so this was a happy solution to our problem.

³This is in contrast to the Scheme designers, who consistently used prefix application of functions and binary operators (for example, `(+ x y)`), instead of adopting mathematical convention.

²The same is true of Miranda users.

(Sections did introduce one problem though: Recall that Haskell has only one prefix operator, namely negation. So the question arises, what is the meaning of `(-42)`? The answer is negative 42! In order to get the function `\x-> x-42` one must write either `\x-> x-42`, or `(subtract 42)`, where `subtract` is a predefined function in Haskell. This “problem” with sections was viewed more as a problem with prefix operators, but as mentioned earlier the committee decided not to buck convention in its treatment of negation.)

Once we had sections, and in particular a way to convert infix operators into ordinary functional values, we then asked ourselves why we couldn’t go the other way. Could we design a mechanism to convert an ordinary function into an infix operator? Our simple solution was to enclose a function identifier in backquotes. For example, `x ‘f’ y` is the same as `f x y`. We liked the generality that this afforded, as well as the ability to use “words” as infix operators. For example, we felt that list membership, say, was more readable when written as `x ‘elem’ xs` rather than `elem x xs`. Miranda used a similar notation, `x $elem xs`, taken from Art Evans’ PAL (Evans, 1968).

4.3 Namespaces and keywords

Namespaces were a point of considerable discussion in the Haskell Committee. We wanted the user to have as much freedom as possible, while avoiding any form of ambiguity. So we carefully defined a set of lexemes for each namespace that were *orthogonal* when they needed to be, and *overlapped* when context was sufficient to distinguish their meaning. As an example of orthogonality, we designed normal variables, infix operators, normal data constructors, and infix data constructors to be mutually exclusive. As an example of overlap, capitalised names can, in the same lexical scope, refer to a type constructor, a data constructor, *and* a module, since whenever the name `Foo` appears, it is clear from context to which entity it is referring. For example, it is quite common to declare a single-constructor data type like this:

```
data Vector = Vector Float Float
```

Here, `Vector` is the name of the data type, and the name of the single data constructor of that type.

We adopted from Miranda the convention that data constructors are capitalised while variables are not, and added a similar convention for infix constructors, which in Haskell must start with a colon. The latter convention was chosen for consistency with our use (adopted from SASL, KRC, and Miranda) of a single colon `:` for the list “`cons`” operator. (The choice of `:` for `cons` and `::` for type signatures, by the way, was a hotly contested issue (ML does the opposite) and remains controversial to this day.)

As a final comment, a small contingent of the Haskell Committee argued that shadowing of variables should *not* be allowed, because introducing a shadowed name might accidentally capture a variable bound in an outer scope. But outlawing shadowing is inconsistent with alpha renaming—it means that you must know the bound names of the inner scope in order to choose a name for use in an outer scope. So, in the end, Haskell allowed shadowing.

Haskell has 21 reserved keywords that cannot be used as names for values or types. This is a relatively low number (Erlang has 28, OCaml has 48, Java has 50, C++ has 63—and Miranda has only 10), and keeping it low was a priority of the Haskell Committee. Also, we tried hard to avoid keywords (such as “`as`”) that might otherwise be useful variable names.

4.4 Declaration style vs. expression style

As our discussions evolved, it became clear that there were two different styles in which functional programs could be written: “*declaration style*” and “*expression style*”. For example, here is the filter function written in both styles⁴:

```
filter :: (a -> Bool) -> [a] -> [a]

-- Declaration style
filter p [] = []
filter p (x:xs) | p x = x : rest
                | otherwise = rest
                where
                    rest = filter p xs

-- Expression style
filter = \p -> \xs ->
    case xs of
        [] -> []
        (x:xs) -> let
            rest = filter p xs
            in if (p x)
                then x : rest
                else rest
```

The declaration style attempts, so far as possible, to define a function by multiple equations, each of which uses pattern matching and/or guards to identify the cases it covers. In contrast, in the expression style a function is built up by composing expressions together to make bigger expressions. Each style is characterised by a set of syntactic constructs:

Declaration style	Expression-style
where clause	let expression
Function arguments on left hand side	Lambda abstraction
Pattern matching in function definitions	case expression
Guards on function definitions	if expression

The declaration style was heavily emphasised in Turner’s languages KRC (which introduced guards for the first time) and Miranda (which introduced a `where` clause scoping over several guarded equations, *including the guards*). The expression style dominates in other functional languages, such as Lisp, ML, and Scheme.

It took some while to identify the stylistic choice as we have done here, but once we had done so, we engaged in furious debate about which style was “better.” An underlying assumption was that if possible there should be “just one way to do something,” so that, for example, having both `let` and `where` would be redundant and confusing.

In the end, we abandoned the underlying assumption, and provided full syntactic support for both styles. This may seem like a classic committee decision, but it is one that the present authors believe was a fine choice, and that we now regard as a strength of the language. Different constructs have different nuances, and real programmers do in practice employ both `let` and `where`, both guards and conditionals, both pattern-matching definitions and `case` expressions—not only in the same program but sometimes in the same function definition. It is certainly true that the additional syntactic sugar makes the language seem more elaborate, but it is a superficial sort of complexity, easily explained by purely syntactic transformations.

⁴The example is a little contrived. One might argue that the code would be less cluttered (in both cases) if one eliminated the `let` or `where`, replacing `rest` with `filter p xs`.

Two small but important matters concern guards. First, Miranda placed guards on the far right-hand side of equations, thus resembling common notation used in mathematics, thus:

```
gcd x y = x,           if x=y
= gcd (x-y) y,        if x>y
= gcd x (y-x),       otherwise
```

However, as mentioned earlier in the discussion of layout, the Haskell Committee did not buy into the idea that programmers should write (or feel forced to write) *short* function definitions, and placing the guard on the far right of a *long* definition seemed like a bad idea. So, we moved them to the left-hand side of the definition (see `filter` and `f` above), which had the added benefit of placing the guard right next to the patterns on formal parameters (which logically made more sense), and in a place more suggestive of the evaluation order (which builds the right operational intuitions). Because of this, we viewed our design as an improvement over conventional mathematical notation.

Second, Haskell adopted from Miranda the idea that a `where` clause is attached to a *declaration*, not an expression, and scopes over the guards as well as the right-hand sides of the declarations. For example, in Haskell one can write:

```
firstSat :: (a->Bool) -> [a] -> Maybe a
firstSat p xs | null xsps = Nothing
               | otherwise = Just xp
               where
                 xsps = filter p xs
                 xp = head xsps
```

Here, `xps` is used in a guard as well as in the binding for `xp`. In contrast, a `let` binding is attached to an *expression*, as can be seen in the second definition of `filter` near the beginning of this subsection. Note also that `xp` is defined only in the second clause—but that is fine since the bindings in the `where` clause are lazy.

4.5 List comprehensions

List comprehensions provide a very convenient notation for maps, filters, and Cartesian products. For example,

```
[ x*x | x <- xs ]
```

returns the squares of the numbers in the list `xs`, and

```
[ f | f <- [1..n], n `mod` f == 0 ]
```

returns a list of the factors of `n`, and

```
concatMap :: (a -> [b]) -> [a] -> [b]
concatMap f xs = [ y | x <- xs, y <- f x ]
```

applies a function `f` to each element of a list `xs`, and concatenates the resulting lists. Notice that each element `x` chosen from `xs` is used to generate a new list (`f x`) for the second generator.

The list comprehension notation was first suggested by John Darlington when he was a student of Rod Burstall. The notation was popularised—and generalised to lazy lists—by David Turner’s use of it in KRC, where it was called a “ZF expression” (named after Zermelo-Fraenkel set theory). Turner put this notation to effective use in his paper “The semantic elegance of applicative languages” (Turner, 1981). Wadler introduced the name “list comprehension” in his paper “How to replace failure by a list of successes” (Wadler, 1985).

For some reason, list comprehensions seem to be more popular in lazy languages; for example they are found in Miranda and Haskell, but not in SML or Scheme. However, they are present in Erlang and more recently have been added to Python, and there are plans to add them to Javascript as array comprehensions.

4.6 Comments

Comments provoked much discussion among the committee, and Wadler later formulated a law to describe how effort was allotted to various topics: semantics is discussed half as much as syntax, syntax is discussed half as much as lexical syntax, and lexical syntax is discussed half as much as the syntax of comments. This was an exaggeration: a review of the mail archives shows that well over half of the discussion concerned semantics, and infix operators and layout provoked more discussion than comments. Still, it accurately reflected that committee members held strong views on low-level details.

Originally, Haskell supported two commenting styles. Depending on your view, this was either a typical committee decision, or a valid response to a disparate set of needs. Short comments begin with a double dash `--` and end with a newline; while longer comments begin with `{-` and end with `-}`, and can be nested. The longer form was designed to make it easy to comment out segments of code, including code containing comments.

Later, Haskell added support for a third convention, literate comments, which first appeared in OL at the suggestion of Richard Bird. (Literate comments also were later adopted by Miranda.) Bird, inspired by Knuth’s work on “literate programming” (Knuth, 1984), proposed reversing the usual comment convention: lines of *code*, rather than lines of *comment*, should be the ones requiring a special mark. Lines that were not comments were indicated by a greater-than sign `>` to the left. For obvious reasons, these non-comment indicators came to be called ‘Bird tracks’.

Haskell later supported a second style of literate comment, where code was marked by `\begin{code}` and `\end{code}` as it is in Latex, so that the same file could serve both as source for a typeset paper and as an executable program.

5. Data types and pattern matching

Data types and pattern matching are fundamental to most modern functional languages (with the notable exception of Scheme). The inclusion of basic algebraic types was straightforward, but interesting issues arose for pattern matching, abstract types, tuples, new types, records, `n+k` patterns, and views.

The style of writing functional programs as a sequence of equations with pattern matching over algebraic types goes back at least to Burstall’s work on structural induction (Burstall, 1969), and his work with his student Darlington on program transformation (Burstall and Darlington, 1977).

Algebraic types as a programming language feature first appeared in Burstall’s NPL (Burstall, 1977) and Burstall, MacQueen, and Sannella’s Hope (Burstall et al., 1980). They were absent from the original ML (Gordon et al., 1979) and KRC (Turner, 1982), but appeared in their successors Standard ML (Milner et al., 1997) and Miranda (Turner, 1986). Equations with conditional guards were introduced by Turner in KRC (Turner, 1982).

5.1 Algebraic types

Here is a simple declaration of an algebraic data type and a function accepting an argument of the type that illustrates the basic features of algebraic data types in Haskell.

```
data Maybe a = Nothing | Just a
```

```
mapMaybe :: (a->b) -> Maybe a -> Maybe b
mapMaybe f (Just x) = Just (f x)
mapMaybe f Nothing = Nothing
```

The `data` declaration declares `Maybe` to be a data type, with two *data constructors* `Nothing` and `Just`. The values of the `Maybe` type take one of two forms: either `Nothing` or `(Just x)`. Data constructors can be used both in *pattern-matching*, to decompose a value of `Maybe` type, and in an *expression*, to build a value of `Maybe` type. Both are illustrated in the definition of `mapMaybe`.

The use of pattern matching against algebraic data types greatly increases readability. Here is another example, this time defining a recursive data type of trees:

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)

size          :: Tree a -> Int
size (Leaf x) = 1
size (Branch t u) = size t + size u + 1
```

Haskell took from Miranda the notion of defining algebraic types as a ‘sum of products’. In the above, a tree is either a leaf or a branch (a sum with two alternatives), a leaf contains a value (a trivial product with only one field), and a branch contains a left and right subtree (a product with two fields). In contrast, Hope and Standard ML separated sums (algebraic types) and products (tuple types); in the equivalent definition of a tree, a branch would take one argument which was itself a tuple of two trees.

In general, an algebraic type specifies a sum of one or more alternatives, where each alternative is a product of zero or more fields. It might have been useful to permit a sum of zero alternatives, which would be a completely empty type, but at the time the value of such a type was not appreciated.

Haskell also took from Miranda the rule that constructor names always begin with a capital, making it easy to distinguish constructors (like `Leaf` and `Branch`) from variables (like `x`, `t`, and `u`). In Standard ML, it is common to use lower case for both; if a pattern consists of a single identifier it can be hard to tell whether this is a variable (which will match anything) or a constructor with no arguments (which matches only that constructor).

Haskell further extended this rule to apply to type constructors (like `Tree`) and type variables (like `a`). This uniform rule was unusual. In Standard ML type variables were distinguished by starting with a tick (e.g., `tree `a`), and in Miranda type variables were written as a sequence of one or more asterisks (e.g., `tree *`).

5.2 Pattern matching

The semantics of pattern matching in lazy languages is more complex than in strict languages, because laziness means that whether one chooses to first match against a variable (doesn’t force evaluation) or a constructor (does force evaluation) can change the semantics of a program, in particular, whether or not the program terminates.

In SASL, KRC, Hope, SML, and Miranda, matching against equations is in order from top to bottom, with the first matching equation being used. Moreover in SASL, KRC, and Miranda, matching is from left to right within each left-hand-side—which is important in a lazy language, since as soon as a non-matching pattern is found, matching proceeds to the next equation, potentially avoiding non-termination or an error in a match further to the right. Eventually, these choices were made for Haskell as well, after considering at length and rejecting some other possibilities:

- Tightest match, as used in Hope+ (Field et al., 1992).
- Sequential equations, as introduced by Huet and Levy (Huet and Levy, 1979).
- Uniform patterns, as described by Wadler in Chapter 5 of Peyton Jones’s textbook (Peyton Jones, 1987).

Top-to-bottom, left-to-right matching was simple to implement, fit nicely with guards, and offered greater expressiveness compared to the other alternatives. But the other alternatives had a semantics in which the order of equations did not matter, which aids equational reasoning (see (Hudak, 1989) for more details). In the end, it was thought better to adopt the more widely used top-to-bottom design than to choose something that programmers might find limiting.

5.3 Abstract types

In Miranda, abstract data types were supported by a special language construct, `abstype`:

```
abstype stack * == [*]
with push :: * -> stack * -> stack *
      pop :: stack * -> *
      empty :: stack *
      top :: stack * -> *
      isEmpty :: stack * -> bool
push x xs = x:xs
pop (x:xs) = xs
empty = []
top (x:xs) = x
isEmpty xs = xs = []
```

Here the types `stack *` and `[*]` are synonyms within the definitions of the named functions, but distinguished everywhere else.

In Haskell, instead of a special construct, the module system is used to support data abstraction. One constructs an abstract data type by introducing an algebraic type, and then exporting the type but hiding its constructors. Here is an example:

```
module Stack( Stack, push, pop,
              empty, top, isEmpty ) where
data Stack a = Stk [a]
push x (Stk xs) = Stk (x:xs)
pop (Stk (x:xs)) = Stk xs
empty = Stk []
top (Stk (x:xs)) = x
isEmpty (Stk xs) = null xs
```

Since the constructor for the data type `Stack` is hidden (the export list would say `Stack(Stk)` if it were exposed), outside of this module a stack can only be built from the operations `push`, `pop`, and `empty`, and examined with `top` and `isempty`.

Haskell’s solution is somewhat cluttered by the `Stk` constructors, but in exchange an extra construct is avoided, and the types of the operations can be inferred if desired. The most important point is that Haskell’s solution allows one to give a different instance to a type-class for the abstract type than for its representation:

```
instance Show Stack where
  show s = ...
```

The `Show` instance for `Stack` can be different from the `Show` instance for lists, and there is no ambiguity about whether a given subexpression is a `Stack` or a list. It was unclear to us how to achieve this effect with `abstype`.

5.4 Tuples and irrefutable patterns

An expression that diverges (or calls Haskell’s `error` function) is considered to have the value “bottom”, usually written \perp , a value that belongs to every type. There is an interesting choice to be made about the semantics of tuples: are \perp and (\perp, \perp) distinct values? In the jargon of denotational semantics, a *lifted* tuple semantics distinguishes the two values, while an *unlifted* semantics treats them as the same value.

In an implementation, the two values will be *represented* differently, but under the unlifted semantics they must be indistinguishable to the programmer. The only way in which they might be distinguished is by pattern matching; for example:

```
f (x,y) = True
```

If this pattern match evaluates *f*'s argument then $f \perp = \perp$, but $f(\perp, \perp) = \text{True}$, thereby distinguishing the two values. One can instead consider this definition to be equivalent to

```
f t = True
where
  x = fst t
  y = snd t
```

in which case $f \perp = \text{True}$ and the two values are indistinguishable.

This apparently arcane semantic point became a subject of great controversy in the Haskell Committee. Miranda's design identified \perp with (\perp, \perp) , which influenced us considerably. Furthermore, this identification made currying an exact isomorphism:

```
(a,b) -> c ≈ a -> b -> c
```

But there were a number of difficulties. For a start, should single-constructor data types, such as

```
data Pair a b = Pair a b
```

share the same properties as tuples, with a semantic discontinuity induced by adding a second constructor? We were also concerned about the efficiency of this lazy form of pattern matching, and the space leaks that might result. Lastly, the unlifted form of tuples is essentially incompatible with *seq*—another controversial feature of the language, discussed in Section 10.3—because parallel evaluation would be required to implement *seq* on unlifted tuples.

In the end, we decided to make both tuples and algebraic data types have a lifted semantics, so that pattern matching always induces evaluation. However, in a somewhat uneasy compromise, we also reintroduced lazy pattern-matching, in the form of tilde-patterns, thus:

```
g :: Bool -> (Int,Int) -> Int
g b ~(x,y) = if b then x+y else 0
```

The tilde “ \sim ” makes matching lazy, so that the pattern match for (x,y) is performed only if x or y is demanded; that is, in this example, when b is *True*. Furthermore, pattern matching in *let* and *where* clauses is always lazy, so that *g* can also be written:

```
g x pr = if b then x+y else 0
where
  (x,y) = pr
```

(This difference in the semantics of pattern matching between *let*/*where* and *case*/ λ can perhaps be considered a wart on the language design—certainly it complicates the language description.) All of this works uniformly when there is more than one constructor in the data type:

```
h :: Bool -> Maybe Int -> Int
h b ~(Just x) = if b then x else 0
```

Here again, *h* evaluates its second argument only if b is *True*.

5.5 Newtype

The same choice described above for tuples arose for any algebraic type with one constructor. In this case, just as with tuples, there was a choice as to whether or not the semantics should be lifted. From Haskell 1.0, it was decided that algebraic types with a single constructor should have a lifted semantics. From Haskell 1.3 onwards

there was also a second way to introduce a new algebraic type with a single constructor and a single component, with an unlifted semantics. The main motivation for introducing this had to do with abstract data types. It was unfortunate that the Haskell definition of *Stack* given above forced the representation of stacks to be not quite isomorphic to lists, as lifting added a new bottom value \perp distinct from *Stk* \perp . Now one could avoid this problem by replacing the data declaration in *Stack* above with the following declaration.

```
newtype Stack a = Stk [a]
```

We can view this as a way to define a new type isomorphic to an existing one.

5.6 Records

One of the most obvious omissions from early versions of Haskell was the absence of *records*, offering named fields. Given that records are extremely useful in practice, why were they omitted?

The strongest reason seems to have been that there was no obvious “right” design. There are a huge number of record systems, variously supporting record extension, concatenation, update, and polymorphism. All of them have a complicating effect on the type system (e.g., row polymorphism and/or subtyping), which was already complicated enough. This extra complexity seemed particularly undesirable as we became aware that type classes could be used to encode at least some of the power of records.

By the time the Haskell 1.3 design was under way, in 1993, the user pressure for named fields in data structures was strong, so the committee eventually adopted a minimalist design originally suggested by Mark Jones: record syntax in Haskell 1.3 (and subsequently) is simply syntactic sugar for equivalent operation on regular algebraic data types. Neither record-polymorphic operations nor subtyping are supported.

This minimal design has left the field open for more sophisticated proposals, of which the best documented is TRex (Gaster and Jones, 1996) (Section 6.7). New record proposals continue to appear regularly on the Haskell mailing list, along with ingenious ways of encoding records using type classes (Kiselyov et al., 2004).

5.7 n+k patterns

An algebraic type isomorphic to the natural numbers can be defined as follows:

```
data Nat = Zero | Succ Nat
```

This definition has the advantage that one can use pattern matching in definitions, but the disadvantage that the unary representation implied in the definition is far less efficient than the built-in representation of integers. Instead, Haskell provides so-called *n+k* patterns that provide the benefits of pattern matching without the loss of efficiency. (The *n+k* pattern feature can be considered a special case of a *view* (Wadler, 1987) (see Section 5.8) combined with convenient syntax.) Here is an example:

```
fib :: Int -> Int
fib 0      = 1
fib 1      = 1
fib (n+2)  = fib n + fib (n+1)
```

The pattern *n+k* only matches a value m if $m \geq k$, and if it succeeds it binds *n* to $m - k$.

Patterns of the form *n+k* were suggested for Haskell by Wadler, who first saw them in Gödel's incompleteness proof (Gödel, 1931), the core of which is a proof-checker for logic, coded using recursive equations in a style that would seem not unfamiliar to users

of Haskell. They were earlier incorporated into Darlington's NPL (Burstall and Darlington, 1977), and (partially at Wadler's instigation) into Miranda.

This seemingly innocuous bit of syntax provoked a great deal of controversy. Some users considered $n+k$ patterns essential, because they allowed function definition by cases over the natural numbers (as in `fib` above). But others worried that the `Int` type did not, in fact, denote the natural numbers. Indeed, worse was to come: since in Haskell the numeric literals (0, 1 etc) were overloaded, it seemed only consistent that `fib`'s type should be

```
fib :: Num a => a -> a
```

although the programmer is, as always, allowed to specify a less general type, such as `Int -> Int` above. In Haskell, one can perfectly well apply `fib` to matrices! This gave rise to a substantial increase in the complexity of pattern matching, which now had to invoke overloaded comparison and arithmetic operations. Even syntactic niceties resulted:

```
n + 1 = 7
```

is a (function) definition of `+`, while

```
(n + 1) = 7
```

is a (pattern) definition of `n`—so apparently redundant brackets change the meaning completely!

Indeed, these complications led to the majority of the Haskell Committee suggesting that $n+k$ patterns be removed. One of the very few bits of horse-trading in the design of Haskell occurred when Hudak, then Editor of the Report, tried to convince Wadler to agree to remove $n+k$ patterns. Wadler said he would agree to their removal only if some other feature went (we no longer remember which). In the end, $n+k$ patterns stayed.

5.8 Views

Wadler had noticed there was a tension between the convenience of pattern matching and the advantages of data abstraction, and suggested *views* as a programming language feature that lessens this tension. A view specifies an isomorphism between two data types, where the second must be algebraic, and then permits constructors of the second type to appear in patterns that match against the first (Wadler, 1987). Several variations on this initial proposal have been suggested, and Chris Okasaki (Okasaki, 1998b) provides an excellent review of these.

The original design of Haskell included views, and was based on the notion that the constructors and views exported by a module should be indistinguishable. This led to complications in export lists and derived type classes, and by April 1989 Wadler was arguing that the language could be simplified by removing views.

At the time views were removed, Peyton Jones wanted to add views to an experimental extension of Haskell, and a detailed proposal to include views in Haskell 1.3 was put forward by Burton and others (Burton et al., 1996). But views never made it back into the language nor appeared among the many extensions available in some implementations.

There is some talk of including views or similar features in Haskell', a successor to Haskell now under discussion, but they are unlikely to be included as they do not satisfy the criterion of being “tried and true”.

6. Haskell as a type-system laboratory

Aside from laziness, type classes are undoubtedly Haskell's most distinctive feature. They were originally proposed early in the design process, by Wadler and Blott (Wadler and Blott, 1989), as a

principled solution to a relatively small problem (operator overloading for numeric operations and equality). As time went on, type classes began to be generalised in a variety of interesting and surprising ways, some of them summarised in a 1997 paper “Type classes: exploring the design space” (Peyton Jones et al., 1997).

An entirely unforeseen development—perhaps encouraged by type classes—is that Haskell has become a kind of laboratory in which numerous type-system extensions have been designed, implemented, and applied. Examples include polymorphic recursion, higher-kinded quantification, higher-rank types, lexically scoped type variables, generic programming, template meta-programming, and more besides. The rest of this section summarises the historical development of the main ideas in Haskell's type system, beginning with type classes.

6.1 Type classes

The basic idea of type classes is simple enough. Consider equality, for example. In Haskell we may write

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool

instance Eq Int where
  i1 == i2 = eqInt i1 i2
  i1 /= i2 = not (i1 == i2)

instance (Eq a) => Eq [a] where
  [] == [] = True
  (x:xs) == (y:ys) = (x == y) && (xs == ys)
  xs /= ys = not (xs == ys)

member :: Eq a => a -> [a] -> Bool
member x [] = False
member x (y:ys) | x==y = True
                 | otherwise = member x ys
```

In the instance for `Eq Int`, we assume that `eqInt` is a primitive function defining equality at type `Int`. The type signature for `member` uses a form of bounded quantification: it declares that `member` has type `a -> [a] -> Bool`, for any type `a` that is an instance of the class `Eq`. A `class` declaration specifies the methods of the class (just two in this case, namely `(==)` and `(/=)`) and their types. A type is made into an instance of the class using an `instance` declaration, which provides an implementation for each of the class's methods, at the appropriate instance type.

A particularly attractive feature of type classes is that they can be translated into so-called “dictionary-passing style” by a type-directed transformation. Here is the translation of the above code:

```
data Eq a = MkEq (a->a->Bool) (a->a->Bool)
eq (MkEq e _) = e
ne (MkEq _ n) = n

dEqInt :: Eq Int
dEqInt = MkEq eqInt (\x y -> not (eqInt x y))
dEqList :: Eq a -> Eq [a]
dEqList d = MkEq el (\x y -> not (el x y))
  where el [] [] = True
        el (x:xs) (y:ys) = eq d x y && el xs ys
        el _ _ = False

member :: Eq a -> a -> [a] -> Bool
member d x [] = False
member d x (y:ys) | eq d x y = True
                  | otherwise = member d x ys
```

The class declaration translates to a data type declaration, which declares a *dictionary* for Eq, that is, a record of its methods. The functions eq and ne select the equality and inequality method from this dictionary. The member function takes a dictionary parameter of type Eq a, corresponding to the Eq a constraint in its original type, and performs the membership test by extracting the equality method from this dictionary using eq. Finally, an instance declaration translates to a function that takes some dictionaries and returns a more complicated one. For example, dEqList takes a dictionary for Eq a and returns a dictionary for Eq [a].

Once type classes were adopted as part of the language design, they were immediately applied to support the following main groups of operations: equality (Eq) and ordering (Ord); converting values to and from strings (Read and Show); enumerations (Enum); numeric operations (Num, Real, Integral, Fractional, Floating, RealFrac and RealFloat); and array indexing (Ix). The rather daunting collection of type classes used to categorise the numeric operations reflected a slightly uneasy compromise between algebraic purity (which suggested many more classes, such as Ring and Monoid) and pragmatism (which suggested fewer).

In most statically typed languages, the type system checks consistency, but one can understand how the program will *execute* without considering the types. Not so in Haskell: the dynamic semantics of the program necessarily depends on the way that its type-class overloading is resolved by the type checker. Type classes have proved to be a very powerful and convenient mechanism but, because more is happening “behind the scenes”, it is more difficult for the programmer to reason about what is going to happen.

Type classes were extremely serendipitous: they were invented at exactly the right moment to catch the imagination of the Haskell Committee, and the fact that the very first release of Haskell had thirteen type classes in its standard library indicates how rapidly they became pervasive. But beyond that, they led to a wildly richer set of opportunities than their initial purpose, as we discuss in the rest of this section.

6.2 The monomorphism restriction

A major source of controversy in the early stages was the so-called “monomorphism restriction.” Suppose that genericLength has this overloaded type:

```
genericLength :: Num a => [b] -> a
```

Now consider this definition:

```
f xs = (len, len)
      where
        len = genericLength xs
```

It looks as if len should be computed only once, but it can actually be computed *twice*. Why? Because we can infer the type len :: (Num a) => a; when desugared with the dictionary-passing translation, len becomes a *function* that is called once for each occurrence of len, each of which might be used at a different type.

Hughes argued strongly that it was unacceptable to silently duplicate computation in this way. His argument was motivated by a program he had written that ran exponentially slower than he expected. (This was admittedly with a very simple compiler, but we were reluctant to make performance differences as big as this dependent on compiler optimisations.)

Following much debate, the committee adopted the now-notorious monomorphism restriction. Stated briefly, it says that a definition that does not look like a function (i.e. has no arguments on the left-hand side) should be monomorphic in any overloaded type

variables. In this example, the rule forces len to be used at the same type at both its occurrences, which solves the performance problem. The programmer can supply an explicit type signature for len if polymorphic behaviour is required.

The monomorphism restriction is manifestly a wart on the language. It seems to bite every new Haskell programmer by giving rise to an unexpected or obscure error message. There has been much discussion of alternatives. The Glasgow Haskell Compiler (GHC, Section 9.1) provides a flag:

```
-fno-monomorphism-restriction
```

to suppress the restriction altogether. But in all this time, no truly satisfactory alternative has evolved.

6.3 Ambiguity and type defaulting

We rapidly discovered a second source of difficulty with type classes, namely *ambiguity*. Consider the following classic example:

```
show :: Show a => a -> String
read :: Read a => String -> a

f :: String -> String
f s = show (read s)
```

Here, show converts a value of any type in class Show to a String, while read does the reverse for any type in class Read. So f appears well-typed... but the difficulty is there is nothing to specify the type of the intermediate subexpression (read s). Should read parse an Int from s, or a Float, or even a value of type Maybe Int? There is nothing to say which should be chosen, and the choice affects the semantics of the program. Programs like this are said to be *ambiguous* and are rejected by the compiler. The programmer may then say which types to use by adding a type signature, thus:

```
f :: String -> String
f s = show (read s :: Int)
```

However, sometimes rejecting the un-annotated program seems unacceptably pedantic. For example, consider the expression

```
(show (negate 4))
```

In Haskell, the literal 4 is short for (fromInteger (4 :: Integer)), and the types of the functions involved are as follows:

```
fromInteger :: Num a => Integer -> a
negate :: Num a => a -> a
show :: Show a => a -> String
```

Again the expression is ambiguous, because it is not clear whether the computation should be done at type Int, or Float, or indeed any other numeric type. Performing numerical calculations on constants is one of the very first things a Haskell programmer does, and furthermore there is more reason to expect numeric operations to behave in similar ways for different types than there is for non-numeric operations. After much debate, we compromised by adding an *ad hoc* rule for choosing a particular default type. When at least one of the ambiguous constraints is numeric but all the constraints involve only classes from the Standard Prelude, then the constrained type variable is *defaultable*. The programmer may specify a list of types in a special top-level default declaration, and these types are tried, in order, until one satisfies all the constraints.

This rule is clumsy but conservative: it tries to avoid making an arbitrary choice in all but a few tightly constrained situations. In fact, it seems *too* conservative for Haskell interpreters. Notably,

consider the expression (show []). Are we trying to show a list of Char or a list of Int, or what? Of course, it does not matter, since the result is the same in all cases, but there is no way for the type system to know that. GHC therefore relaxes the defaulting rules further for its interactive version GHCi.

6.4 Higher-kinded polymorphism

The first major, unanticipated development in the type-class story came when Mark Jones, then at Yale, suggested parameterising a class over a type *constructor* instead of over a *type*, an idea he called *constructor classes* (Jones, 1993). The most immediate and persuasive application of this idea was to monads (discussed in Section 7), thus:

```
class Monad m where
  return :: a -> m a
  (">>=) :: m a -> (a -> m b) -> m b
```

Here, the type variable *m* has kind⁵ *→*, so that the Monad class can be instantiated at a type constructor. For example, this declaration makes the Maybe type an instance of Monad by instantiating *m* with Maybe, which has kind *→*:

```
data Maybe a = Nothing | Just a
```

```
instance Monad Maybe where
  return x      = Just x
  Nothing >>= k = Nothing
  Just x >>= k = k x
```

So, for example, instantiating return's type (*a* → *m a*) with *m*=Maybe gives the type (*a* → Maybe *a*), and that is indeed the type of the return function in the instance declaration.

Jones's paper appeared in 1993, the same year that monads became popular for I/O (Section 7). The fact that type classes so directly supported monads made monads far more accessible and popular; and dually, the usefulness of monadic I/O ensured the adoption of higher-kinded polymorphism. However, higher-kinded polymorphism has independent utility: it is entirely possible, and occasionally very useful, to declare data types parameterised over higher kinds, such as:

```
data ListFunctor f a = Nil | Cons a (f a)
```

Furthermore, one may need functions quantified over higher-kinded type variables to process nested data types (Okasaki, 1999; Bird and Paterson, 1999).

Type inference for a system involving higher kinds seems at first to require higher-order unification, which is both much harder than traditional first-order unification and lacks most general unifiers (Huet, 1975). However, by treating higher-kinded type constructors as uninterpreted functions and not allowing lambda at the type level, Jones's paper (Jones, 1993) shows that ordinary first-order unification suffices. The solution is a little *ad hoc*—for example, the order of type parameters in a data-type declaration can matter—but it has an excellent power-to-weight ratio. In retrospect, higher-kinded quantification is a simple, elegant, and useful generalisation of the conventional Hindley-Milner typing discipline (Milner, 1978). All this was solidified into the Haskell 1.3 Report, which was published in 1996.

6.5 Multi-parameter type classes

While Wadler and Blott's initial proposal focused on type classes with a single parameter, they also observed that type classes might

⁵ Kinds classify types just as types classify values. The kind * is pronounced “type”, so if *m* has kind *→*, then *m* is a type-level function mapping one type to another.

be generalised to multiple parameters. They gave the following example:

```
class Coerce a b where
  coerce :: a -> b

instance Coerce Int Float where
  coerce = convertIntToFloat
```

Whereas a single-parameter type class can be viewed as a predicate over types (for example, Eq *a* holds whenever *a* is a type for which equality is defined), a multi-parameter class can be viewed a relation between types (for example, Coerce *a* *b* holds whenever *a* is a subtype of *b*).

Multi-parameter type classes were discussed in several early papers on type classes (Jones, 1991; Jones, 1992; Chen et al., 1992), and they were implemented in Jones's language Gofer (see Section 9.3) in its first 1991 release. The Haskell Committee was resistant to including them, however. We felt that single-parameter type classes were already a big step beyond our initial conservative design goals, and they solved the problem we initially addressed (overloading equality and numeric operations). Going beyond that would be an unforced step into the dark, and we were anxious about questions of overlap, confluence, and decidability of type inference. While it was easy to define coerce as above, it was less clear when type inference would make it usable in practice. As a result, Haskell 98 retained the single-parameter restriction.

As time went on, however, user pressure grew to adopt multi-parameter type classes, and GHC adopted them in 1997 (version 3.00). However, multi-parameter type classes did not really come into their own until the advent of functional dependencies.

6.6 Functional dependencies

The trouble with multi-parameter type classes is that it is very easy to write ambiguous types. For example, consider the following attempt to generalise the Num class:

```
class Add a b r where
  (+) :: a -> b -> r

instance Add Int Int Int where ...
instance Add Int Float Float where ...
instance Add Float Int Float where ...
instance Add Float Float Float where ...
```

Here we allow the programmer to add numbers of different types, choosing the result type based on the input types. Alas, even trivial programs have ambiguous types. For example, consider:

```
n = x + y
```

where *x* and *y* have type Int. The difficulty is that the compiler has no way to figure out the type of *n*. The programmer intended that if the arguments of (+) are both Int then so is the result, but that intent is implied only by the absence of an instance declaration such as

```
instance Add Int Int Float where ...
```

In 2000, Mark Jones published “Type classes with functional dependencies”, which solves the problem (Jones, 2000). The idea is to borrow a technique from the database community and declare an explicit functional dependency between the parameters of a class, thus:

```
class Add a b r | a b -> r where ...
```

The “*a* *b* → *r*” says that fixing *a* and *b* should fix *r*, resolving the ambiguity.

But that was not all. The combination of multi-parameter classes and functional dependencies turned out to allow computation at the type level. For example:

```

data Z = Z
data S a = S a

class Sum a b r | a b -> r

instance Sum Z b b
instance Sum a b r => Sum (S a) b (S r)

```

Here, `Sum` is a three-parameter class with no operations. The relation `Sum ta tb tc` holds if the type `tc` is the Peano representation (at the type level) of the sum of `ta` and `tb`. By liberalising other Haskell 98 restrictions on the form of instance declarations (and perhaps thereby risking non-termination in the type checker), it turned out that one could write arbitrary computations at the type level, in logic-programming style. This realisation gave rise to an entire cottage industry of type-level programming that shows no sign of abating (e.g., Hallgren, 2001; McBride, 2002; Kiselyov et al., 2004), as well as much traffic on the Haskell mailing list. It also led to a series of papers suggesting more direct ways of expressing such programs (Neubauer et al., 2001; Neubauer et al., 2002; Chakravarty et al., 2005b; Chakravarty et al., 2005a).

Jones's original paper gave only an informal description of functional dependencies, but (as usual with Haskell) that did not stop them from being implemented and widely used. These applications have pushed functional dependencies well beyond their motivating application. Despite their apparent simplicity, functional dependencies have turned out to be extremely tricky in detail, especially when combined with other extensions such as local universal and existential quantification (Section 6.7). Efforts to understand and formalise the design space are still in progress (Glynn et al., 2000; Sulzmann et al., 2007).

6.7 Beyond type classes

As if all this were not enough, type classes have spawned numerous variants and extensions (Peyton Jones et al., 1997; Lämmel and Peyton Jones, 2005; Shields and Peyton Jones, 2001). Furthermore, even leaving type classes aside, Haskell has turned out to be a setting in which advanced type systems can be explored and applied. The rest of this section gives a series of examples; space precludes a proper treatment of any of them, but we give citations for the interested reader to follow up.

Existential data constructors A useful programming pattern is to package up a value with functions over that value and existentially quantify the package (Mitchell and Plotkin, 1985). Perry showed in his dissertation (Perry, 1991b; Perry, 1991a) and in his implementation of Hope+ that this pattern could be expressed with almost no new language complexity, simply by allowing a data constructor to mention type variables in its arguments that do not appear in its result. For example, in GHC one can say this:

```

data T = forall a. MkT a (a->Int)
f :: T -> Int
f (MkT x g) = g x

```

Here the constructor `MkT` has type $\forall a. a \rightarrow (a \rightarrow \text{Int}) \rightarrow T$; note the occurrence of `a` in the argument type but not the result. A value of type `T` is a package of a value of some (existentially quantified) type τ , and a function of type $\tau \rightarrow \text{Int}$. The package can be unpacked with ordinary pattern matching, as shown in the definition of `f`.

This simple but powerful idea was later formalised by Odersky and Läufer (Läufer and Odersky, 1994). Läufer also described how

to integrate existentials with Haskell type classes (Läufer, 1996). This extension was first implemented in hbc and is now a widely used extension of Haskell 98: every current Haskell implementation supports the extension.

Extensible records Mark Jones showed that type classes were an example of a more general framework he called *qualified types* (Jones, 1994). With his student Benedict Gaster he developed a second instance of the qualified-type idea, a system of polymorphic, extensible records called TRex (Gaster and Jones, 1996; Gaster, 1998). The type qualification in this case is a collection of *lacks predicates*, thus:

```

f :: (r\x, r\y)
=> Rec (x:Int, y:Int | r) -> Int
f p = (#x p) + (#y p)

```

The type should be read as follows: `f` takes an argument record with an `x` and `y` fields, plus other fields described by the row-variable `r`, and returns an `Int`. The *lacks* predicate `(r\x, r\y)` says that `r` should range only over rows that do not have an `x` or `y` field—otherwise the argument type `Rec (x:Int, y:Int | r)` would be ill formed. The selector `#x` selects the `x` field from its argument, so `(#x p)` is what would more traditionally be written `p.x`. The system can accommodate a full complement of polymorphic operations: selection, restriction, extension, update, and field renaming (although not concatenation).

Just as each type-class constraint corresponds to a runtime argument (a dictionary), so each *lacks* predicate is also witnessed by a runtime argument. The witness for the predicate `(r\1)` is the offset in `r` at which a field labelled 1 would be inserted. Thus `f` receives extra arguments that tell it where to find the fields it needs. The idea of passing extra arguments to record-polymorphic functions is not new (Ohori, 1995), but the integration with a more general framework of qualified types is particularly elegant; the reader may find a detailed comparison in Gaster's dissertation (Gaster, 1998).

Implicit parameters A third instantiation of the qualified-type framework, so-called “implicit parameters”, was developed by Lewis, Shields, Meijer, and Launchbury (Lewis et al., 2000). Suppose you want to write a pretty-printing library that is parameterised by the page width. Then each function in the library must take the page width as an extra argument, and in turn pass it to the functions it calls:

```

pretty :: Int -> Doc -> String
pretty pw doc = if width doc > pw
                 then pretty2 pw doc
                 else pretty3 pw doc

```

These extra parameters are quite tiresome, especially when they are only passed on unchanged. Implicit parameters arrange that this parameter passing happens implicitly, rather like dictionary passing, thus:

```

pretty :: (?pw:Int) => Doc -> String
pretty doc = if width doc > ?pw
              then pretty2 doc
              else pretty3 doc

```

The explicit parameter turns into an implicit-parameter type constraint; a reference to the page width itself is signalled by `?pw`; and the calls to `pretty2` and `pretty3` no longer pass an explicit `pw` parameter (it is passed implicitly instead). One way of understanding implicit parameters is that they allow the programmer to make selective use of dynamic (rather than lexical) scoping. (See (Kiselyov and Shan, 2004) for another fascinating approach to the problem of distributing configuration information such as the page width.)

Polymorphic recursion This feature allows a function to be used polymorphically in its own definition. It is hard to *infer* the type of such a function, but easy to *check* that the definition is well typed, given the type signature of the function. So Haskell 98 allows polymorphic recursion when (and only when) the programmer explicitly specifies the type signature of the function. This innovation is extremely simple to describe and implement, and sometimes turns out to be essential, for example when using nested data types (Bird and Paterson, 1999).

Higher-rank types Once one starts to use polymorphic recursion, it is not long before one encounters the need to abstract over a polymorphic function. Here is an example inspired by (Okasaki, 1999):

```
type Sq v a = v (v a)    -- Square matrix:
                        -- A vector of vectors

sq_index :: (forall a . Int -> v a -> a)
           -> Int -> Int -> Sq v a -> a
sq_index index i j m = index i (index j m)
```

The function `index` is used inside `sq_index` at two different types, so it must be polymorphic. Hence the first argument to `sq_index` is a polymorphic function, and `sq_index` has a so-called rank-2 type. In the absence of any type annotations, higher-rank types make type inference undecidable; but a few explicit type annotations from the programmer (such as that for `sq_index` above) transform the type inference problem into an easy one (Peyton Jones et al., 2007).

Higher-rank types were first implemented in GHC in 2000, in a rather *ad hoc* manner. At that time there were two main motivations: one was to allow data constructors with polymorphic fields, and the other was to allow the `runST` function to be defined (Launchbury and Peyton Jones, 1995). However, once implemented, another cottage industry sprang up offering examples of their usefulness in practice (Baars and Swierstra, 2002; Lämmel and Peyton Jones, 2003; Hinze, 2000; Hinze, 2001), and GHC's implementation has become much more systematic and general (Peyton Jones et al., 2007).

Generalised algebraic data types GADTs are a simple but far-reaching generalisation of ordinary algebraic data types (Section 5). The idea is to allow a data constructor's return type to be specified directly:

```
data Term a where
  Lit :: Int -> Term Int
  Pair :: Term a -> Term b -> Term (a,b)
  ...etc..
```

In a function that performs pattern matching on `Term`, the pattern match gives *type* as well as *value* information. For example, consider this function:

```
eval :: Term a -> a
eval (Lit i)      = i
eval (Pair a b) = (eval a, eval b)
...
```

If the argument matches `Lit`, it must have been built with a `Lit` constructor, so `a` must be `Int`, and hence we may return `i` (an `Int`) in the right-hand side. This idea is very well known in the type-theory community (Dybjer, 1991). Its advent in the world of programming languages (under various names) is more recent, but it seems to have many applications, including generic programming, modelling programming languages, maintaining invariants in data structures (e.g., red-black trees), expressing constraints in domain-specific embedded languages (e.g. security constraints), and modelling objects (Hinze, 2003; Xi et al., 2003; Cheney and Hinze,

2003; Sheard and Pasalic, 2004; Sheard, 2004). Type inference for GADTs is somewhat tricky, but is now becoming better understood (Pottier and Régis-Gianas, 2006; Peyton Jones et al., 2004), and support for GADTs was added to GHC in 2005.

Lexically scoped type variables In Haskell 98, it is sometimes impossible to write a type signature for a function, because type signatures are always *closed*. For example:

```
prefix :: a -> [[a]] -> [[a]]
prefix x yss = map xcons yss
where
  xcons :: [a] -> [a] -- BAD!
  xcons ys = x : ys
```

The type signature for `xcons` is treated by Haskell 98 as specifying the type $\forall a. [a] \rightarrow [a]$, and so the program is rejected. To fix the problem, some kind of lexically scoped type variables are required, so that “`a`” is bound by `prefix` and used in the signature for `xcons`. In retrospect, the omission of lexically scoped type variables was a mistake, because polymorphic recursion and (more recently) higher-rank types absolutely require type signatures. Interestingly, though, scoped type variables were not omitted after fierce debate; on the contrary, they were barely discussed; we simply never realised how important type signatures would prove to be.

There are no great technical difficulties here, although there is an interesting space of design choices (Milner et al., 1997; Meijer and Claessen, 1997; Shields and Peyton Jones, 2002; Sulzmann, 2003).

Generic programming A *generic* function behaves in a uniform way on arguments of any data types, while having a few type-specific cases. An example might be a function that capitalises all the strings that are in a big data structure: the generic behaviour is to traverse the structure, while the type-specific case is for strings. In another unforeseen development, Haskell has served as the host language for a remarkable variety of experiments in generic programming, including: approaches that use pure Haskell 98 (Hinze, 2004); ones that require higher-rank types (Lämmel and Peyton Jones, 2003; Lämmel and Peyton Jones, 2005); ones that require a more specific language extension, such as PolyP (Jansson and Jeuring, 1997), and derivable type classes (Hinze and Peyton Jones, 2000); and whole new language designs, such as Generic Haskell (Löh et al., 2003). See (Hinze et al., 2006) for a recent survey of this active research area.

Template meta-programming Inspired by the template meta-programming of C++ and the staged type system of MetaML (Taha and Sheard, 1997), GHC supports a form of type-safe meta-programming (Sheard and Peyton Jones, 2002).

6.8 Summary

Haskell's type system has developed extremely anarchically. Many of the new features described above were sketched, implemented, and applied well before they were formalised. This anarchy, which would be unthinkable in the Standard ML community, has both strengths and weaknesses. The strength is that the design space is explored much more quickly, and tricky corners are often (but not always!) exposed. The weakness is that the end result is extremely complex, and programs are sometimes reduced to experiments to see what will and will not be acceptable to the compiler.

Some notable attempts have been made to bring order to this chaos. Karl-Filip Faxen wrote a static semantics for the whole of Haskell 98 (Faxen, 2002). Mark Jones, who played a prominent role in several of these developments, developed a theory of *qualified types*, of which type classes, implicit parameters, and extensible records are all instances (Jones, 1994; Jones, 1995). More recently, he wrote

a paper giving the complete code for a Haskell 98 type inference engine, which is a different way to formalise the system (Jones, 1999). Martin Sulzmann and his colleagues have applied the theory of *constraint-handling rules* to give a rich framework to reason about type classes (Sulzmann, 2006), including the subtleties of functional dependencies (Glynn et al., 2000; Sulzmann et al., 2007).

These works do indeed nail down some of the details, but the result is still dauntingly complicated. The authors of the present paper have the sense that we are still awaiting a unifying insight that will not only explain but also simplify the chaotic world of type classes, without throwing the baby out with the bath water.

Meanwhile, it is worth asking *why* Haskell has proved so friendly a host language for type-system innovation. The following reasons seem to us to have been important. On the technical side:

- The purity of the language removed a significant technical obstacle to many type-system innovations, namely dealing with mutable state.
- Type classes, and their generalisation to qualified types (Jones, 1994), provided a rich (albeit rather complex) framework into which a number of innovations fitted neatly; examples include extensible records and implicit parameters.
- Polymorphic recursion was in the language, so the idea that every legal program should typecheck without type annotations (a tenet of ML) had already been abandoned. This opens the door to features for which unaided inference is infeasible.

But there were also nontechnical factors at work:

- The Haskell Committee encouraged innovation right from the beginning and, far from exercising control over the language, disbanded itself in 1999 (Section 3.7).
- The two most widely used implementations (GHC, Hugs) both had teams that encouraged experimentation.
- Haskell has a smallish, and rather geeky, user base. New features are welcomed, and even breaking changes are accepted.

7. Monads and input/output

Aside from type classes (discussed in Section 6), *monads* are one of the most distinctive language design features in Haskell. Monads were not in the original Haskell design, because when Haskell was born a “monad” was an obscure feature of category theory whose implications for programming were largely unrecognised. In this section we describe the symbiotic evolution of Haskell’s support for input/output on the one hand, and monads on the other.

7.1 Streams and continuations

The story begins with I/O. The Haskell Committee was resolute in its decision to keep the language pure—meaning no side effects—so the design of the I/O system was an important issue. We did not want to lose expressive power just because we were “pure,” since interfacing to the real world was an important pragmatic concern. Our greatest fear was that Haskell would be viewed as a toy language because we did a poor job addressing this important capability.

At the time, the two leading contenders for a solution to this problem were *streams* and *continuations*. Both were understood well enough theoretically, both seemed to offer considerable expressiveness, and both were certainly pure. In working out the details of these approaches, we realised that in fact they were functionally equivalent—that is, it was possible to completely model stream I/O with continuations, and vice versa. Thus in the Haskell 1.0 Report,

we first defined I/O in terms of streams, but also included a completely equivalent design based on continuations.

It is worth mentioning that a third model for I/O was also discussed, in which the state of the world is passed around and updated, much as one would pass around and update any other data structure in a pure functional language. This “world-passing” model was never a serious contender for Haskell, however, because we saw no easy way to ensure “single-threaded” access to the world state. (The Clean designers eventually solved this problem through the use of “uniqueness types” (Achten and Plasmeijer, 1995; Barendsen and Smetsers, 1996).) In any case, all three designs were considered, and Hudak and his student Sundaresan wrote a report describing them, comparing their expressiveness, and giving translations between them during these deliberations (Hudak and Sundaresan, 1989). In this section we give a detailed account of the stream-based and continuation-based models of I/O, and follow in Section 7.2 with the monadic model of I/O that was adopted for Haskell 1.3 in 1996.

Stream-based I/O Using the stream-based model of purely functional I/O, used by both Ponder and Miranda, a program is represented as a value of type:

```
type Behaviour = [Response] -> [Request]
```

The idea is that a program generates a Request to the operating system, and the operating system reacts with some Response. Lazy evaluation allows a program to generate a request prior to processing any responses. A suitably rich set of Requests and Responses yields a suitably expressive I/O system. Here is a partial definition of the Request and Response data types as defined in Haskell 1.0:

```
data Request = ReadFile Name
             | WriteFile Name String
             | AppendFile Name String
             | DeleteFile Name
             | ...
data Response = Success
              | Str String
              | Failure IOError
              | ...
type Name = String
```

As an example, Figure 3 presents a program, taken from the Haskell 1.0 Report, that prompts the user for the name of a file, echoes the filename as typed by the user, and then looks up and displays the contents of the file on the standard output. Note the reliance on lazy patterns (indicated by ~) to assure that the response is not “looked at” prior to the generation of the request.

With this treatment of I/O there was no need for any special-purpose I/O syntax or I/O constructs. The I/O system was defined entirely in terms of how the operating system interpreted a program having the above type—that is, it was defined in terms of what response the OS generated for each request. An abstract specification of this behaviour was defined in the Appendix of the Haskell 1.0 Report, by giving a definition of the operating system as a function that took as input an initial state and a collection of Haskell programs and used a single nondeterministic merge operator to capture the parallel evaluation of the multiple Haskell programs.

Continuation-based I/O Using the continuation-based model of I/O, a program was still represented as a value of type Behaviour, but instead of having the user manipulate the requests and responses directly, a collection of *transactions* were defined that cap-

tured the effect of each request/response pair in a continuation-passing style. Transactions were just functions. For each request (a constructor, such as `ReadFile`) there corresponded a transaction (a function, such as `readFile`).

The request `ReadFile` name induced either a failure response “`Failure msg`” or success response “`Str contents`” (see above). So the corresponding transaction `readFile` name accepted two continuations, one for failure and one for success.

```
type Behaviour = [Response] -> [Request]
type FailCont = IOError -> Behaviour
type StrCont = String -> Behaviour
```

One can define this transaction in terms of streams as follows.

```
readFile :: Name -> FailCont -> StrCont -> Behaviour
readFile name fail succ ~(resp:resps) =
  = ReadFile name :
    case resp of
      Str val      -> succ val resps
      Failure msg -> fail msg resps
```

If the transaction failed, the failure continuation would be applied to the error message; if it succeeded, the success continuation would be applied to the contents of the file. In a similar way, it is straightforward to define each of the continuation-based transactions in terms of the stream-based model of I/O.

Using this style of I/O, the example given earlier in stream-based I/O can be rewritten as shown in Figure 4. The code uses the standard failure continuation, `abort`, and an auxiliary function `let`. The use of a function called `let` reflects the fact that `let` expressions were not in Haskell 1.0! (They appeared in Haskell 1.1.)

Although the two examples look somewhat similar, the continuation style was preferred by most programmers, since the flow of control was more localised. In particular, the pattern matching required by stream-based I/O forces the reader’s focus to jump back and forth between the patterns (representing the responses) and the requests.

Above we take streams as primitive and define continuations in terms of them. Conversely, with some cleverness it is also possible to take continuations as primitive and define streams in terms of them (see (Hudak and Sundaresan, 1989), where the definition of streams in terms of continuations is attributed to Peyton Jones). However, the definition of streams in terms of continuations was inefficient, requiring linear space and quadratic time in terms of the number of requests issued, as opposed to the expected constant space and linear time. For this reason, Haskell 1.0 defined streams as primitive, and continuations in terms of them, even though continuations were considered easier to use for most purposes.

7.2 Monads

We now pause the story of I/O while we bring *monads* onto the scene. In 1989, Eugenio Moggi published at LICS a paper on the use of monads from category theory to describe features of programming languages, which immediately attracted a great deal of attention (Moggi, 1989; Moggi, 1991). Moggi used monads to modularise the structure of a denotational semantics, systematising the treatment of diverse features such as state and exceptions. But a denotational semantics can be viewed as an interpreter written in a functional language. Wadler recognised that the technique Moggi had used to structure semantics could be fruitfully applied to structure other functional programs (Wadler, 1992a; Wadler, 1992b). In effect, Wadler used monads to *express* the same programming language features that Moggi used monads to *describe*.

For example, say that you want to write a program to rename every occurrence of a bound variable in a data structure representing a lambda expression. This requires some way to generate a fresh name every time a bound variable is encountered. In ML, you would probably introduce a reference cell that contains a count, and increment this count each time a fresh name is required. In Haskell, lacking reference cells, you would probably arrange that each function that must generate fresh names accepts an old value of the counter and returns an updated value of the counter. This is straightforward but tedious, and errors are easily introduced by misspelling one of the names used to pass the current count in to or out of a function application. Using a *state transformer* monad would let you hide all the “plumbing.” The monad itself would be responsible for passing counter values, so there is no chance to misspell the associated names.

A monad consists of a type constructor `M` and a pair of functions, `return` and `>>=` (sometimes pronounced “bind”). Here are their types:

```
return :: a -> M a
(>>=) :: M a -> (a -> M b) -> M b
```

One should read “`M a`” as the type of a *computation* that returns a value of type `a` (and perhaps performs some side effects). Say that `m` is an expression of type `M a` and `n` is an expression of type `M b` with a free variable `x` of type `a`. Then the expression

```
m >>= (\x -> n)
```

has type `M b`. This performs the computation indicated by `m`, binds the value returned to `x`, and performs the computation indicated by `n`. It is analogous to the expression

```
let x = m in n
```

in a language with side effects such as ML, except that the types do not indicate the presence of the effects: in the ML version, `m` has type `a` instead of `M a`, and `n` has type `b` instead of `M b`. Further, monads give quite a bit of freedom in how one defines the operators `return` and `>>=`, while ML fixes a single built-in notion of computation and sequencing.

Here are a few examples of the notions of side effects that one can define with monads:

- A *state transformer* is used to thread state through a program. Here `M a` is `ST s a`, where `s` is the state type.

```
type ST s a = s -> (a,s)
```

A state transformer is a function that takes the old state (of type `s`) and returns a value (of type `a`) and the new state (of type `s`). For instance, to thread a counter through a program we might take `s` to be integer.

- A *state reader* is a simplified state transformer. It accepts a state that the computation may depend upon, but the computation never changes the state. Here `M a` is `SR s a`, where `s` is the state type.

```
type SR s a = s -> a
```

- An *exception* monad either returns a value or raises an exception. Here `M a` is `Exc e a`, where `e` is the type of the error message.

```
data Exc e a = Exception e | OK a
```

- A *continuation* monad accepts a continuation. Here `M a` is `Cont r a`, where `r` is the result type of the continuation.

```
type Cont r a = (a -> r) -> r
```

```

main :: Behaviour
main ~(Success : ~((Str userInput) : ~(Success : ~(r4 : _))))
= [ AppendChan stdout "enter filename\n",
  ReadChan stdin,
  AppendChan stdout name,
  ReadFile name,
  AppendChan stdout
    (case r4 of
      Str contents    -> contents
      Failure ioerr  -> "can't open file")
] where (name : _) = lines userInput

```

Figure 3. Stream-based I/O

```

main :: Behaviour
main = appendChan stdout "enter filename\n" abort (
  readChan stdin abort          (\userInput ->
    letE (lines userInput)      (\(name : _) ->
      appendChan stdout name abort
      readFile name fail        (
        (\contents ->
          appendChan stdout contents abort done)))))
where
  fail ioerr = appendChan stdout "can't open file" abort done

abort     :: FailCont
abort err resp = []

letE      :: a -> (a -> b) -> b
letE x k  =  k x

```

Figure 4. Continuation I/O

```

main :: IO ()
main = appendChan stdout "enter filename\n"  >>
  readChan stdin           >>= \userInput ->
  let (name : _) = lines userInput in
  appendChan stdout name   >>
  catch (readFile name >>= \contents ->
    appendChan stdout contents)
    (appendChan stdout "can't open file")

```

Figure 5. Monadic I/O

```

main :: IO ()
main = do appendChan stdout "enter filename\n"
  userInput <- readChan stdin
  let (name : _) = lines userInput
  appendChan stdout name
  catch (do contents <- readFile name
    appendChan stdout contents)
    (appendChan stdout "can't open file")

```

Figure 6. Monadic I/O using do notation

- A *list* monad can be used to model nondeterministic computations, which return a sequence of values. Here $M\ a$ is *List* a , which is just the type of lists of values of type a .

```
type List a = [a]
```

- A *parser* monad can be used to model parsers. The input is the string to be parsed, and the result is list of possible parses, each consisting of the value parsed and the remaining unparsed string. It can be viewed as a combination of the state transformer monad (where the state is the string being parsed) and the list monad (to return each possible parse in turn). Here $M\ a$ is *Parser* a .

```
type Parser a = String -> [(a, String)]
```

Each of the above monads has corresponding definitions of `return` and `>>=`. There are three laws that these definitions should satisfy in order to be a true monad in the sense defined by category theory. These laws guarantee that composition of functions with side effects is *associative* and has an *identity* (Wadler, 1992b). For example, the latter law is this:

$$\text{return } x \gg= f = f x$$

Each of the monads above has definitions of `return` and `>>=` that satisfy these laws, although Haskell provides no mechanism to ensure this. Indeed, in practice some Haskell programmers use the monadic types and programming patterns in situations where the monad laws do not hold.

A monad is a kind of “programming pattern”. It turned out that this pattern can be directly expressed in Haskell, using a type class, as we saw earlier in Section 6.4:

```
class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b
```

The *Monad* class gives concrete expression to the mathematical idea that any type constructor that has suitably typed unit and bind operators is a monad. That concrete expression has direct practical utility, because we can now write useful monadic combinators that will work for *any* monad. For example:

```
sequence :: Monad m => [m a] -> m [a]
sequence []      = return []
sequence (m:ms) = m >>= \x ->
                  sequence ms >>= \xs ->
                  return (x:xs)
```

The intellectual reuse of the idea of a monad is directly reflected in actual code reuse in Haskell. Indeed, there are whole Haskell libraries of monadic functions that work for *any* monad. This happy conjunction of monads and type classes gave the two a symbiotic relationship: each made the other much more attractive.

Monads turned out to be very helpful in structuring quite a few functional programs. For example, GHC’s type checker uses a monad that combines a state transformer (representing the current substitution used by the unifier), an exception monad (to indicate an error if some type failed to unify), and a state reader monad (to pass around the current program location, used when reporting an error). Monads are often used in combination, as this example suggests, and by abstracting one level further one can build *monad transformers* in Haskell (Steele, 1993; Liang et al., 1995; Harrison and Kamin, 1998). The Liang, Hudak, and Jones paper was the first to show that a modular interpreter could be written in Haskell using monad transformers, but it required type class extensions supported only in Gofer (an early Haskell interpreter—see Section 9). This was one of the examples that motivated a flurry of extensions to type classes (see Section 6) and to the development of the monad

transformer library. Despite the utility of monad transformers, monads do not compose in a nice, modular way, a research problem that is still open (Jones and Duponcheel, 1994; Lüth and Ghani, 2002).

Two different forms of syntactic sugar for monads appeared in Haskell at different times. Haskell 1.3 adopted Jones’s “do-notation,” which was itself derived from John Launchbury’s paper on lazy imperative programming (Launchbury, 1993). Subsequently, Haskell 1.4 supported “monad comprehensions” as well as do-notation (Wadler, 1990a)—an interesting reversal, since the comprehension notation was proposed before do-notation! Most users preferred the do-notation, and generalising comprehensions to monads meant that errors in ordinary list comprehensions could be difficult for novices to understand, so monad comprehensions were removed in Haskell 98.

7.3 Monadic I/O

Although Wadler’s development of Moggi’s ideas was not directed towards the question of input/output, he and others at Glasgow soon realised that monads provided an ideal framework for I/O. The key idea is to treat a value of type $IO\ a$ as a “computation” that, when performed, might perform input and output before delivering a value of type a . For example, `readFile` can be given the type

```
readFile :: Name -> IO String
```

So `readFile` is a function that takes a `Name` and returns a computation that, when performed, reads the file and returns its contents as a `String`.

Figure 5 shows our example program rewritten using monads in two forms. It makes use of the monad operators `>>=`, `return`, `>>`, and `catch`, which we discuss next. The first two are exactly as described in the previous section, but specialised for the IO monad. So `return x` is the trivial computation of type $IO\ a$ (where $x : a$) that performs no input or output and returns the value x . Similarly, $(>>=)$ is sequential composition; $(m >>= k)$ is a computation that, when performed, performs m , applies k to the result to yield a computation, which it then performs. The operator $(>>)$ is sequential composition when we want to discard the result of the first computation:

```
(>>) :: IO a -> IO b -> IO b
m >> n = m >>= \_ -> n
```

The Haskell IO monad also supports *exceptions*, offering two new primitives:

```
ioError :: IOError -> IO a
catch   :: IO a -> (IOError -> IO a) -> IO a
```

The computation `(ioError e)` fails, throwing exception e . The computation `(catch m h)` runs computation m ; if it succeeds, then its result is the result of the `catch`; but if it fails, the exception is caught and passed to h .

The same example program is shown once more, rewritten using Haskell’s do-notation, in Figure 6. This notation makes (the monadic parts of) Haskell programs appear much more imperative!

Haskell’s input/output interface is *specified* monadically. It can be *implemented* using continuations, thus:

```
type IO a = FailCont -> SuccCont a -> Behaviour
```

(The reader may like to write implementations of `return`, `(>>=)`, `catch` and so on, using this definition of IO .) However, it is also possible to implement the IO monad in a completely different style, without any recourse to a stream of requests and responses. The implementation in GHC uses the following one:

```
type IO a = World -> (a, World)
```

An `IO` computation is a function that (logically) takes the state of the world, and returns a modified world as well as the return value. Of course, GHC does not actually pass the world around; instead, it passes a dummy “token,” to ensure proper sequencing of actions in the presence of lazy evaluation, and performs input and output as actual side effects! Peyton Jones and Wadler dubbed the result “imperative functional programming” (Peyton Jones and Wadler, 1993).

The monadic approach rapidly dominated earlier models. The types are more compact, and more informative. For example, in the continuation model we had

```
readFile :: Name -> FailCont -> StrCont -> Behaviour
```

The type is cluttered with success and failure continuations (which must be passed by the programmer) and fails to show that the result is a `String`. Furthermore, the types of `IO` computations could be polymorphic:

```
readIORef :: IORRef a -> IO a
writeIORef :: IORRef a -> a -> IO ()
```

These types cannot be written with a fixed `Request` and `Response` type. However, the big advantage is conceptual. It is much easier to think abstractly in terms of computations than concretely in terms of the details of failure and success continuations. The monad abstracts away from these details, and makes it easy to change them in future. The reader may find a tutorial introduction to the `IO` monad, together with various further developments in (Peyton Jones, 2001).

Syntax matters An interesting syntactic issue is worth pointing out in the context of the development of Haskell’s I/O system. Note in the continuation example in Figure 4 the plethora of parentheses that tend to pile up as lambda expressions become nested. Since this style of programming was probably going to be fairly common, the Haskell Committee decided quite late in the design process to change the precedence rules for lambda in the context of infix operators, so that the continuation example could be written as follows:

```
main :: Behaviour
main = appendChan stdout "enter filename\n" >>>
    readChan stdin           >>> \ userInput ->
    let (name : _) = lines userInput in
    appendChan stdout name >>>
    readFile name fail      (\ contents ->
        appendChan stdout contents abort done)
where
    fail ioerr = appendChan stdout "can't open file"
                           abort done
```

where `f >>> x = f` `abort x`. Note the striking similarity of this code to the monadic code in Figure 5. It can be made even more similar by defining a suitable `catch` function, although doing so would be somewhat pedantic.

Although these two code fragments have a somewhat imperative feel because of the way they are laid out, it was really the advent of `do`-notation—not monads themselves—that made Haskell programs look more like conventional imperative programs (for better or worse). This syntax seriously blurred the line between purely functional programs and imperative programs, yet was heartily adopted by the Haskell Committee. In retrospect it is worth asking whether this same (or similar) syntactic device could have been used to make stream or continuation-based I/O look more natural.

7.4 Subsequent developments

Once the `IO` monad was established, it was rapidly developed in various ways that were not part of Haskell 98 (Peyton Jones, 2001). Some of the main ones are listed below.

Mutable state. From the very beginning it was clear that the `IO` monad could also support mutable locations and arrays (Peyton Jones and Wadler, 1993), using these monadic operations:

```
newIORef   :: a -> IO (IORRef a)
readIORef  :: IORRef a -> IO a
writeIORef :: IORRef a -> a -> IO ()
```

An exciting and entirely unexpected development was Launchbury and Peyton Jones’s discovery that imperative computations could be securely encapsulated inside a pure function. The idea was to parameterise a state monad with a type parameter `s` that “infected” the references that could be generated in that monad:

```
newSTRef   :: a -> ST s (STRef s a)
readSTRef  :: STRef s a -> ST s a
writeSTRef :: STRef s a -> a -> ST s ()
```

The encapsulation was performed by a single constant, `runST`, with a rank-2 type (Section 6.7):

```
runST :: (forall s. ST s a) -> a
```

A proof based on parametricity ensures that no references can “leak” from one encapsulated computation to another (Launchbury and Peyton Jones, 1995). For the first time this offered the ability to implement a function using an imperative algorithm, with a solid guarantee that no side effects could accidentally leak. The idea was subsequently extended to accommodate block-structured regions (Launchbury and Sabry, 1997), and reused to support encapsulated continuations (Dybvig et al., 2005).

Random numbers need a seed, and the Haskell 98 `Random` library uses the `IO` monad as a source of such seeds.

Concurrent Haskell (Peyton Jones et al., 1996) extends the `IO` monad with the ability to fork lightweight threads, each of which can perform I/O by itself (so that the language semantics becomes, by design, nondeterministic). Threads can communicate with each other using synchronised, mutable locations called `MVars`, which were themselves inspired by the `M`-structures of Id (Barth et al., 1991).

Transactional memory. The trouble with `MVars` is that programs built using them are not *composable*; that is, it is difficult to build big, correct programs by gluing small correct subprograms together, a problem that is endemic to all concurrent programming technology. *Software transactional memory* is a recent and apparently very promising new approach to this problem, and one that fits particularly beautifully into Haskell (Harris et al., 2005).

Exceptions were built into the `IO` monad from the start—see the use of `catch` above—but Haskell originally only supported a single exception mechanism in purely functional code, namely the function `error`, which was specified as bringing the entire program to a halt. This behaviour is rather inflexible for real applications, which might want to catch, and recover from, calls to `error`, as well as pattern-match failures (which also call `error`). The `IO` monad provides a way to achieve this goal without giving up the simple, deterministic semantics of purely functional code (Peyton Jones et al., 1999).

UnsafePerformIO Almost everyone who starts using Haskell eventually asks “how do I get *out* of the `IO` monad?” Alas,

unlike `runST`, which safely encapsulates an imperative computation, there is no safe way to escape from the `IO` monad. That does not stop programmers from wanting to do it, and occasionally with some good reason, such as printing debug messages, whose order and interleaving is immaterial. All Haskell implementations, blushing slightly, therefore provide:

```
unsafePerformIO :: IO a -> a
```

As its name implies, it is not safe, and its use amounts to a promise by the programmer that it does not matter whether the I/O is performed once, many times, or never; and that its relative order with other I/O is immaterial. Somewhat less obviously, it is possible to use `unsafePerformIO` to completely subvert the type system:

```
cast :: a -> b
cast x = unsafePerformIO
  (do writeIORef r x
      readIORef r   )
where r :: IORef a
  r = unsafePerformIO
    (newIORef (error "urk"))
```

It should probably have an even longer name, to discourage its use by beginners, who often use it unnecessarily.

Arrows are an abstract view of computation with the same flavour as monads, but in a more general setting. Originally proposed by Hughes in 1998 (Hughes, 2000; Paterson, 2003), arrows have found a string of applications in graphical user interfaces (Courtney and Elliott, 2001), reactive programming (Hudak et al., 2003), and polytypic programming (Jansson and Jeuring, 1999). As in the case of monads (only more so), arrow programming is very much easier if syntactic support is provided (Paterson, 2001), and this syntax is treated directly by the type checker.

Underlying all these developments is the realisation that *being explicit about effects is extremely useful*, and this is something that we believe may ultimately be seen as one of Haskell's main impacts on mainstream programming⁶. A good example is the development of transactional memory. In an implementation of transactional memory, every read and write to a mutable location must be logged in some way. Haskell's crude effect system (the `IO` monad) means that almost all memory operations belong to purely functional computations, and hence, by construction, do not need to be logged. That makes Haskell a very natural setting for experiments with transactional memory. And so it proved: although transactional memory had a ten-year history in imperative settings, when Harris, Marlow, Herlihy and Peyton Jones transposed it into the Haskell setting they immediately stumbled on two powerful new composition operators (`retry` and `orElse`) that had lain hidden until then (see (Harris et al., 2005) for details).

8. Haskell in middle age

As Haskell has become more widely used for real applications, more and more attention has been paid to areas that received short shrift from the original designers of the language. These areas are of enormous practical importance, but they have evolved more recently and are still in flux, so we have less historical perspective on them. We therefore content ourselves with a brief overview here, in very rough order of first appearance.

⁶“Effects” is shorthand for “side effects”.

8.1 The Foreign Function Interface

One feature that very many applications need is the ability to call procedures written in some other language from Haskell, and preferably vice versa. Once the `IO` monad was established, a variety of *ad hoc* mechanisms rapidly appeared; for example, GHC's very first release allowed the inclusion of literal C code in monadic procedures, and Hugs had an extensibility mechanism that made it possible to expose C functions as Haskell primitives. The difficulty was that these mechanisms tended to be implementation-specific.

An effort gradually emerged to specify an implementation-independent way for Haskell to call C procedures, and vice versa. This so-called Foreign Function Interface (FFI) treats C as a lowest common denominator: once you can call C you can call practically anything else. This exercise was seen as so valuable that the idea of “Blessed Addenda” emerged, a well-specified Appendix to the Haskell 98 Report that contained precise advice regarding the implementation of a variety of language extensions. The FFI Addendum effort was led by Manuel Chakravarty in the period 2001–2003, and finally resulted in the 30-page publication of Version 1.0 in 2003. In parallel with, and symbiotic with, this standardisation effort were a number of pre-processing tools designed to ease the labour of writing all the `foreign import` declarations required for a large binding; examples include Green Card (Nordin et al., 1997), H/Direct (Finne et al., 1998), and C2Hs (Chakravarty, 1999a) among others.

We have used passive verbs in describing this process (“an effort emerged,” “the exercise was seen as valuable”) because it was different in kind to the original development of the Haskell language. The exercise was open to all, but depended critically on the willingness of one person (in this case Manuel Chakravarty) to drive the process and act as Editor for the specification.

8.2 Modules and packages

Haskell's module system emerged with surprisingly little debate. At the time, the sophisticated ML module system was becoming well established, and one might have anticipated a vigorous debate about whether to adopt it for Haskell. In fact, this debate never really happened. Perhaps no member of the committee was sufficiently familiar with ML's module system to advocate it, or perhaps there was a tacit agreement that the combination of type classes and ML modules was a bridge too far. In any case, we eventually converged on a very simple design: the module system is a namespace control mechanism, nothing more and nothing less. This had the great merit of simplicity and clarity—for example, the module system is specified completely separately from the type system—but, even so, some tricky corners remained unexplored for several years (Diatchki et al., 2002).

In versions 1.0–1.3 of the language, every module was specified by an *interface* as well as an *implementation*. A great deal of discussion took place about the syntax and semantics of interfaces; issues such as the duplication of information between interfaces and implementations, especially when a module re-exports entities defined in one of its imports; whether one can deduce from an interface which module ultimately defines an entity; a tension between what a compiler might want in an interface and what a programmer might want to write; and so on. In the end, Haskell 1.4 completely abandoned interfaces as a formal part of the language; instead interface files were regarded as a possible artifact of separate compilation. As a result, Haskell sadly lacks a formally checked language in which a programmer can advertise the interface that the module supports.

8.2.1 Hierarchical module names

As Haskell became more widely used, the fact that the module name space was completely flat became increasingly irksome; for example, if there are two collection libraries, they cannot both use the module name `Map`.

This motivated an effort led by Malcolm Wallace to specify an extension to Haskell that would allow multi-component hierarchical module names (e.g., `Data.Map`), using a design largely borrowed from Java. This design constituted the second “Blessed Addendum,” consisting of a single page that never moved beyond version 0.0 and “Candidate” status⁷. Nevertheless, it was swiftly implemented by GHC, Hugs, and nhc, and has survived unchanged since.

8.2.2 Packaging and distribution

Modules form a reasonable unit of program *construction*, but not of *distribution*. Developers want to distribute a related group of modules as a “package,” including its documentation, licencing information, details about dependencies on other packages, include files, build information, and much more besides. None of this was part of the Haskell language design.

In 2004, Isaac Jones took up the challenge of leading an effort to specify and implement a system called Cabal that supports the construction and distribution of Haskell packages⁸. Subsequently, David Himmelstrup implemented Hackage, a Cabal package server that enables people to find and download Cabal packages. This is not the place to describe these tools, but the historical perspective is interesting: it has taken more than fifteen years for Haskell to gain enough momentum that these distribution and discovery mechanisms have become important.

8.2.3 Summary

The result of all this evolution is a module system distinguished by its modesty. It does about as little as it is possible for a language to do and still call itself a practical programming tool. Perhaps this was a good choice; it certainly avoids a technically complicated area, as a glance at the literature on ML modules will confirm.

8.3 Libraries

It did not take long for the importance of well-specified and well-implemented libraries to become apparent. The initial Haskell Report included an Appendix defining the Standard Prelude, but by Haskell 1.3 (May 1996) the volume of standard library code had grown to the extent that it was given a separate companion Library Report, alongside the language definition.

The libraries defined as part of Haskell 98 were still fairly modest in scope. Of the 240 pages of the Haskell 98 Language and Libraries Report, 140 are language definition while only 100 define the libraries. But real applications need much richer libraries, and an informal library evolution mechanism began, based around Haskell language implementations. Initially, GHC began to distribute a bundle of libraries called `hslibs` but, driven by user desire for cross-implementation compatibility, the Hugs, GHC and nhc teams began in 2001 to work together on a common, open-source set of libraries that could be shipped with each of their compilers, an effort that continues to this day.

⁷ <http://haskell.org/definition>

⁸ <http://haskell.org/cabal>

Part III

Implementations and Tools

9. Implementations

Haskell is a big language, and it is quite a lot of work to implement. Nevertheless, several implementations are available, and we discuss their development in this section.

9.1 The Glasgow Haskell Compiler

Probably the most fully featured Haskell compiler today is the Glasgow Haskell Compiler (GHC), an open-source project with a liberal BSD-style licence.

GHC was begun in January 1989 at the University of Glasgow, as soon as the initial language design was fixed. The first version of GHC was written in LML by Kevin Hammond, and was essentially a new front end to the Chalmers LML compiler. This prototype started to work in June 1989, just as Peyton Jones arrived in Glasgow to join the burgeoning functional programming group there. The prototype compiler implemented essentially all of Haskell 1.0 including views (later removed), type classes, the deriving mechanism, the full module system, and binary I/O as well as both streams and continuations. It was reasonably robust (with occasional spectacular failures), but the larger Haskell prelude stressed the LML prelude mechanism quite badly, and the added complexity of type classes meant the compiler was quite a lot bigger and slower than the base LML compiler. There were quite a few grumbles about this: most people had 4–8Mbyte workstations at that time, and the compiler used a reasonable amount of that memory (upwards of 2Mbytes!). Partly through experience with this compiler, the Haskell Committee introduced the monomorphism restriction, removed views, and made various other changes to the language.

GHC proper was begun in the autumn of 1989, by a team consisting initially of Cordelia Hall, Will Partain, and Peyton Jones. It was designed from the ground up as a complete implementation of Haskell in Haskell, bootstrapped via the prototype compiler. The only part that was shared with the prototype was the parser, which at that stage was still written in Yacc and C. The first beta release was on 1 April 1991 (the date was no accident), but it was another 18 months before the first full release (version 0.10) was made in December 1992. This version of GHC already supported several extensions to Haskell: monadic I/O (which only made it officially into Haskell in 1996), mutable arrays, unboxed data types (Peyton Jones and Launchbury, 1991), and a novel system for space and time profiling (Sansom and Peyton Jones, 1995). A subsequent release (July 1993) added a strictness analyser.

A big difference from the prototype is that GHC uses a very large data type in its front end that accurately reflects the full glory of Haskell’s syntax. All processing that can generate error messages (notably resolving lexical scopes, and type inference) is performed on this data type. This approach contrasts with the more popular method of first removing syntactic sugar, and only then processing a much smaller language. The GHC approach required us to write a great deal of code (broad, but not deep) to process the many constructors of the syntax tree, but has the huge advantage that the error messages could report exactly what the programmer wrote.

After type checking, the program is desugared into an explicitly typed intermediate language called simply “Core” and then processed by a long sequence of Core-to-Core analyses and optimising transformations. The final Core program is transformed in

the Spineless Tagless G-machine (STG) language (Peyton Jones, 1992), before being translated into C or machine code.

The Core language is extremely small — its data type has only a dozen constructors in total — which makes it easy to write a Core-to-Core transformation or analysis pass. We initially based Core on the lambda calculus but then, wondering how to decorate it with types, we realised in 1992 that a ready-made basis lay to hand, namely Girard’s System $F\omega$ (Girard, 1990); all we needed to do was to add data types, let-expressions, and case expressions. GHC appears to be the first compiler to use System F as a typed intermediate language, although at the time we thought it was such a simple idea that we did not think it worth publishing, except as a small section in (Peyton Jones et al., 1993). Shortly afterwards, the same idea was used independently by Morrisett, Harper and Tarditi at Carnegie Mellon in their TIL compiler (Tarditi et al., 1996). They understood its significance much better than the GHC team, and the whole approach of type-directed compilation subsequently became extremely influential.

Several years later, we added a “Core Lint” typechecker that checked that the output of each pass remained well-typed. If the compiler is correct, this check will always succeed, but it provides a surprisingly strong consistency check—many, perhaps most bugs in the optimiser produce type-incorrect code. Furthermore, catching compiler bugs this way is vastly cheaper than generating incorrect code, running it, getting a segmentation fault, debugging it with gdb, and gradually tracing the problem back to its original cause. Core Lint often nails the error immediately. This consistency checking turned out to be one of the biggest benefits of a typed intermediate language, although it took us a remarkably long time to recognise this fact.

Over the fifteen years of its life so far, GHC has grown a huge number of features. It supports dozens of language extensions (notably in the type system), an interactive read/eval/print interface (GHCI), concurrency (Peyton Jones et al., 1996; Marlow et al., 2004), transactional memory (Harris et al., 2005), Template Haskell (Sheard and Peyton Jones, 2002), support for packages, and much more besides. This makes GHC a dauntingly complex beast to understand and modify and, mainly for that reason, development of the core GHC functionality remains with Peyton Jones and Simon Marlow, who both moved to Microsoft Research in 1997.

9.2 hbc

The hbc compiler was written by Lennart Augustsson, a researcher at Chalmers University whose programming productivity beggars belief. Augustsson writes:

“During the spring of 1990 I was eagerly awaiting the first Haskell compiler, it was supposed to come from Glasgow and be based on the LML compiler. And I waited and waited. After talking to Glasgow people at the LISP & Functional Programming conference in Nice in late June of 1990 Staffan Truvé and I decided that instead of waiting even longer we would write our own Haskell compiler based on the LML compiler.

“For various reasons Truvé couldn’t help in the coding of the compiler, so I ended up spending most of July and August coding, sometimes in an almost trance-like state; my head filled with Haskell to the brim. At the end of August I had a mostly complete implementation of Haskell. I decided that hbc would be a cool name for the compiler since it is Haskell Curry’s initials. (I later learnt that this is the name the Glasgow people wanted for their compiler too. But first come, first served.)

“The first release, 0.99, was on August 21, 1990. The implementation had everything from the report (except for File operations) and

also several extensions, many of which are now in Haskell 98 (e.g., operator sections).

“The testing of the compiler at the time of release was really minimal, but it could compile the Standard Prelude—and the Prelude uses a lot of Haskell features. Speaking of the Prelude I think it’s worth pointing out that Joe Fasel’s prelude code must be about the oldest Haskell code in existence, and large parts of it are still unchanged! The prelude code was also remarkably un-buggy for code that had never been compiled (or even type checked) before hbc came along.

“Concerning the implementation, I only remember two problematic areas: modules and type checking. The export/import of names in modules were different in those days (renaming) and there were many conditions to check to make sure a module was valid. But the big stumbling block was the type checking. It was hard to do. This was way before there were any good papers about how it was supposed to be done.

“After the first release hbc became a test bed for various extensions and new features and it lived an active life for over five years. But since the compiler was written in LML it was more or less doomed to dwindle.”

9.3 Gofer and Hugs⁹

GHC and hbc were both fully fledged compilers, themselves implemented in a functional language, and requiring a good deal of memory and disk space. In August 1991, Mark Jones, then a D.Phil. student at the University of Oxford, released an entirely different implementation called Gofer (short for “GOod For Equational Reasoning”). Gofer was an interpreter, implemented in C, developed on an 8MHz 8086 PC with 640KB of memory, and small enough to fit on a single (360KB) floppy disk.

Jones wrote Gofer as a side project to his D.Phil. studies—indeed, he reports that he did not dare tell his thesis adviser about Gofer until it was essentially finished—to learn more about the implementation of functional programming languages. Over time, however, understanding type classes became a central theme of Jones’ dissertation work (Jones, 1994), and he began to use Gofer as a testbed for his experiments. For example, Gofer included the first implementation of multi-parameter type classes, as originally suggested by Wadler and Blott (Wadler and Blott, 1989) and a regular topic of both conversation and speculation on the Haskell mailing list at the time. Gofer also adopted an interesting variant of Wadler and Blott’s dictionary-passing translation (Section 6.1) that was designed to minimise the construction of dictionaries at run time, to work with multiple parameter type classes, and to provide more accurate principal types. At the same time, however, this resulted in small but significant differences between the Haskell and Gofer type systems, so that some Haskell programs would not work in Gofer, and vice versa.

Moving to take a post-doctoral post at Yale in 1992, Jones continued to develop and maintain Gofer, adding support for constructor classes (Section 6.4) in 1992–93 and producing the first implementation of the do-notation in 1994. Both of these features were subsequently adopted in Haskell 98. By modifying the interpreter’s back end, Jones also developed a Gofer-to-C compiler, and he used this as a basis for the first “dictionary-free” implementation of type classes, using techniques from partial evaluation to specialise away the results of the dictionary-passing translation.

After he left Yale in the summer of 1994, Jones undertook a major rewrite of the Gofer code base, to more closely track the Haskell

⁹ The material in this section was largely written by Mark Jones, the author of Gofer and Hugs.

standard. Briefly christened “Hg” (short for Haskell-gofer), the new system soon acquired the name “Hugs” (for “the Haskell User’s Gofer System”). The main development work was mostly complete by the time Jones started work at the University of Nottingham in October 1994, and he hoped that Hugs would not only appease the critics but also help to put his newly founded research group in Nottingham onto the functional programming map. Always enjoying the opportunity for a pun, Jones worked to complete the first release of the system so that he could announce it on February 14, 1995 with the greeting “Hugs on Valentine’s Day!” The first release of Hugs supported almost all of the features of Haskell 1.2, including Haskell-style type classes, stream-based I/O, a full prelude, derived instances, defaults, overloaded numeric literals, and bignum arithmetic. The most prominent missing feature was the Haskell module system; Hugs 1.0 would parse but otherwise ignore module headers and import declarations.

Meanwhile, at Yale, working from Hugs 1.0 and striving to further close the gap with Haskell, Alastair Reid began modifying Hugs to support the Haskell module system. The results of Reid’s work appeared for the first time in the Yale Hugs0 release in June 1996. Meanwhile, Jones had continued his own independent development of Hugs, leading to an independent release of Hugs 1.3 in August 1996 that provided support for new Haskell 1.3 features such as monadic I/O, the labelled field syntax, newtype declarations, and strictness annotations, as well as adding user interface enhancements such as import chasing.

Even before the release of these two different versions of Hugs, Jones and Reid had started to talk about combining their efforts into a single system. The first joint release, Hugs 1.4, was completed in January 1998, its name reflecting the fact that the Haskell standard had also moved on to a new version by that time. Jones, however, had also been working on a significant overhaul of the Hugs type checker to include experimental support for advanced type system features including rank-2 polymorphism, polymorphic recursion, scoped type variables, existentials, and extensible records, and also to restore the support for multi-parameter type classes that had been eliminated in the transition from Gofer to Hugs. These features were considered too experimental for Hugs 1.4 and were released independently as Hugs 1.3c, which was the last version of Hugs to be released without support for Haskell modules.

It had been a confusing time for Hugs users (and developers!) while there were multiple versions of Hugs under development at the same time. This problem was finally addressed with the release of Hugs 98 in March 1999, which merged the features of the previous Yale and Nottingham releases into a single system. Moreover, as the name suggests, this was the first version of Hugs to support the Haskell 98 standard. In fact Hugs 98 was also the last of the Nottingham and Yale releases of Hugs, as both Jones and Reid moved on to other universities at around that time (Jones to OGI and Reid to Utah).

Hugs development has proceeded at a more gentle pace since the first release of Hugs 98, benefiting in part from the stability provided by the standardisation of Haskell 98. But Hugs development has certainly not stood still, with roughly one new formal release each year. Various maintainers and contributors have worked on Hugs during this period, including Jones and Reid, albeit at a reduced level, as well as Peterson, Andy Gill, Johan Nordlander, Jeff Lewis, Sigbjorn Finne, Ross Paterson, and Dimitry Golubovsky. In addition to fixing bugs, these developers have added support for new features including implicit parameters, functional dependencies, Microsoft’s .NET, an enhanced foreign function interface, hierarchical module names, Unicode characters, and a greatly expanded collection of libraries.

9.4 nhc

The original nhc was developed by Niklas Röjemo when he was a PhD student at Chalmers (Rojemo, 1995a). His motivation from the start was to have a space-efficient compiler (Rojemo, 1995b) that could be bootstrapped in a much smaller memory space than required by systems such as hbc and GHC. Specifically he wanted to bootstrap it on his personal machine which had around 2Mbytes main memory.

To help achieve this space-efficiency he made use during development of the first-generation heap-profiling tools—which had previously been developed at York and used to reveal space-leaks in hbc (Runciman and Wakeling, 1992; Runciman and Wakeling, 1993). Because of this link, Röjemo came to York as a post-doctoral researcher where, in collaboration with Colin Runciman, he devised more advanced heap-profiling methods, and used them to find residual space-inefficiencies in nhc, leading to a still more space-efficient version (Rjemo and Runciman, 1996a).

When Röjemo left York around 1996 he handed nhc over to Runciman’s group, for development and free distribution (with due acknowledgements). Malcolm Wallace, a post-doc at York working on functional programming for embedded systems, became the principal keeper and developer of nhc—he has since released a series of distributed versions, tracking Haskell 98, adding standard foreign-function interface and libraries, and making various improvements (Wallace, 1998).

The nhc system has been host to various further experiments. For example, a continuing strand of work relates to space efficiency (Wallace and Runciman, 1998), and more recently the development of the Hat tools for tracing programs (Wallace et al., 2001). In 2006, the York Haskell Compiler project, yhc, was started to re-engineer nhc.

9.5 Yale Haskell

In the 1980s, prior to the development of Haskell, there was an active research project at Yale involving Scheme and a dialect of Scheme called *T*. Several MS and PhD theses grew out of this work, supervised mostly by Hudak. The *Orbit* compiler, an optimising compiler for *T*, was one of the key results of this effort (Kranz et al., 2004; Kranz et al., 1986).

So once Hudak became actively involved in the design of Haskell, it was only natural to apply Scheme compilation techniques in an implementation of Haskell. However, rather than port the ideas to a stand-alone Haskell compiler, it seemed easier to compile Haskell into Scheme or *T*, and then use a Scheme compiler as a back end. Unfortunately, the *T* compiler was no longer being maintained and had problems with compilation speed. *T* was then abandoned in favour of *Common Lisp* to address performance and portability issues. This resulted in what became known as *Yale Haskell*.

John Peterson and Sandra Loosmore, both Research Scientists at Yale, were the primary implementers of Yale Haskell. To achieve reasonable performance, Yale Haskell used strictness analysis and type information to compile the strict part of Haskell into very efficient Lisp code. The CMU lisp compiler was able to generate very good numeric code from Lisp with appropriate type annotations. The compiler used a dual-entry point approach to allow very efficient first-order function calls. Aggressive in-lining was able to generate code competitive with other languages (Hartel et al., 1996). In addition, Yale Haskell performed various optimisations intended to reduce the overhead of lazy evaluation (Hudak and Young, 1986; Bloss et al., 1988b; Bloss et al., 1988a; Young, 1988; Bloss, 1988).

Although performance was an important aspect of the Yale compiler, the underlying Lisp system allowed the Yale effort to focus attention on the Haskell programming environment. Yale Haskell was the first implementation to support both compiled and interpreted code in the same program (straightforward, since Lisp systems had been doing that for years). It also had a very nice emacs-based programming environment in which simple two-keystroke commands could be used to evaluate expressions, run dialogues, compile modules, turn specific compiler diagnostics on and off, enable and disable various optimisers, and run a tutorial on Haskell. Commands could even be queued, thus allowing, for example, a compilation to run in the background as the editing of a source file continued in emacs in the foreground.

Another nice feature of Yale Haskell was a “scratch pad” that could be automatically created for any module. A scratch pad was a logical extension of a module in which additional function and value definitions could be added, but whose evaluation did not result in recompilation of the module. Yale Haskell also supported many Haskell language extensions at the time, and thus served as an excellent test bed for new ideas. These extensions included monads, dynamic types, polymorphic recursion, strictness annotations, inlining pragmas, specialising over-loaded functions, mutually recursive modules, and a flexible foreign function interface for both C and Common Lisp.

Ultimately, the limitations of basing a Haskell compiler on a Common Lisp back-end caught up with the project. Although early on Yale Haskell was competitive with GHC and other compilers, GHC programs were soon running two to three times faster than Yale Haskell programs. Worse, there was no real hope of making Yale Haskell run any faster without replacing the back-end and runtime system. Optimisations such as reusing the storage in a thunk to hold the result after evaluation were impossible with the Common Lisp runtime system. The imperative nature of Lisp prevented many other optimisations that could be done in a Haskell-specific garbage collector and memory manager. Every thunk introduced an extra level of indirection (a Lisp cons cell) that was unnecessary in the other Haskell implementations. While performance within the strict subset of Haskell was comparable with other systems, there was a factor of 3 to 5 in lazy code that could not be overcome due to the limitations of the Lisp back end. For this reason, in addition to the lack of funding to pursue further research in this direction, the Yale Haskell implementation was abandoned circa 1995.

9.6 Other Haskell compilers

One of the original inspirations for Haskell was the MIT dataflow project, led by Arvind, whose programming language was called Id. In 1993 Arvind and his colleagues decided to adopt Haskell’s syntax and type system, while retaining Id’s eager, parallel evaluation order, I-structures, and M-structures. The resulting language was called pH (short for “parallel Haskell”), and formed the basis of Nikhil and Arvind’s textbook on implicit parallel programming (Nikhil and Arvind, 2001). The idea of evaluating Haskell eagerly rather than lazily (while retaining non-strict semantics), but on a uniprocessor, was also explored by Maessen’s Eager Haskell (Maessen, 2002) and Ennals’s optimistic evaluation (Ennals and Peyton Jones, 2003).

All the compilers described so far were projects begun in the early or mid ’90s, and it had begun to seem that Haskell was such a dauntingly large language that no further implementations would emerge. However, in the last five years several new Haskell implementation projects have been started.

Helium. The Helium compiler, based at Utrecht, is focused especially on teaching, and on giving high-quality type error messages (Heeren et al., 2003b; Heeren et al., 2003a).

UHC and EHC. Utrecht is also host to two other Haskell compiler projects, UHC and EHC (<http://www.cs.uu.nl/wiki/Center/ResearchProjects>).

jhc is a new compiler, developed by John Meacham. It is focused on aggressive optimisation using whole-program analysis. This whole-program approach allows a completely different approach to implementing type classes, without using dictionary-passing. Based on early work by Johnsson and Boquist (Boquist, 1999), jhc uses flow analysis to support a de-functionalised representation of thunks, which can be extremely efficient.

The York Haskell Compiler, yhc, is a new compiler for Haskell 98, based on nhc but with an entirely new back end.

9.7 Programming Environments

Until recently, with the notable exception of Yale Haskell, little attention has been paid by Haskell implementers to the programming environment. That is now beginning to change. Notable examples include the Haskell Refactorer (Li et al., 2003); the GHC Visual Studio plug-in (Visual Haskell), developed by Krasimir Angelov and Simon Marlow (Angelov and Marlow, 2005); and the EclipseFP plug-in for Haskell, developed by Leif Frenzel, Thiago Arrais, and Andrei de A Formiga¹⁰.

10. Profiling and debugging

One of the disadvantages of lazy evaluation is that operational aspects such as evaluation order, or the contents of a snapshot of memory at any particular time, are not easily predictable from the source code—and indeed, can vary between executions of the same code, depending on the demands the context makes on its result. As a result, conventional profiling and debugging methods are hard to apply. We have all *tried* adding side-effecting print calls to record a trace of execution, or printing a backtrace of the stack on errors, only to discover that the information obtained was too hard to interpret to be useful. Developing successful profiling and debugging tools for Haskell has taken considerable research, beginning in the early 1990s.

10.1 Time profiling

At the beginning of the 1990s, Patrick Sansom and Peyton Jones began working on profiling Haskell. The major difficulty was finding a sensible way to assign costs. The conventional approach, of assigning costs to functions and procedures, works poorly for higher-order functions such as `map`. Haskell provides many such functions, which are designed to be reusable in many different contexts and for many different tasks—so these functions feature prominently in time profiles. But knowing that `map` consumes 20% of execution time is little help to the programmer—we need to know instead *which* occurrence of `map` stands for a large fraction of the time. Likewise, when one logical task is implemented by a combination of higher-order functions, then the time devoted to the task is divided among these functions in a way that disguises the time spent on the task itself. Thus a new approach to assigning costs was needed.

The new idea Sansom and Peyton Jones introduced was to label the source code with *cost centres*, either manually (to reflect the programmer’s intuitive decomposition into tasks) or automatically.

¹⁰ <http://eclipsefp.sourceforge.net>

The profiling tool they built then assigned time and space costs to one of these cost centres, thus aggregating all the costs for one logical task into one count (Sansom and Peyton Jones, 1995).

Assigning costs to explicitly labelled cost centres is much more subtle than it sounds. Programmers expect that costs should be assigned to the closest enclosing cost centre—but should this be the closest *lexically* enclosing or the closest *dynamically* enclosing cost centre? (Surprisingly, the best answer is the closest lexically enclosing one (Sansom and Peyton Jones, 1995).) In a language with first-class functions, should the cost of *evaluating* a function necessarily be assigned to the same cost centre as the costs of *calling* the function? In a call-by-need implementation, where the cost of using a value the first time can be much greater than the cost of using it subsequently, how can one ensure that cost assignments are independent of evaluation order (which the programmer should not need to be aware of)? These questions are hard enough to answer that Sansom and Peyton Jones felt the need to develop a formal cost semantics, making the assignment of costs to cost centres precise. This semantics was published at POPL in 1995, but a prototype profiling tool was already in use with GHC in 1992. Not surprisingly, the availability of a profiler led rapidly to faster Haskell programs, in particular speeding up GHC itself by a factor of two.

10.2 Space profiling

Sansom and Peyton Jones focused on profiling *time* costs, but at the same time Colin Runciman and David Wakeling were working on *space*, by profiling the contents of the heap. It had been known for some time that lazy programs could sometimes exhibit astonishingly poor space behaviour—so-called *space leaks*. Indeed, the problem was discussed in Hughes’s dissertation in 1984, along with the selective introduction of strictness to partially fix them, but there was no practical way of *finding* the causes of space leaks in large programs. Runciman and Wakeling developed a profiler that could display a graph of heap contents over time, classified by the function that allocated the data, the top-level constructor of the data, or even combinations of the two (for example, “show the allocating functions of all the cons cells in the heap over the entire program run”). The detailed information now available enabled lazy programmers to make dramatic improvements to space efficiency: as the first case study, Runciman and Wakeling reduced the peak space requirements of a classification program for propositional logic by two orders of magnitude, from 1.3 megabytes to only 10K (Runciman and Wakeling, 1993). Runciman and Wakeling’s original profiler worked for LML, but it was rapidly adopted by Haskell compilers, and the visualisation tool they wrote to display heap profiles is still in use to this day.

By abstracting away from *evaluation order*, lazy evaluation also abstracts away from *object lifetimes*, and that is why lazy evaluation contributes to space leaks. Programmers who cannot predict—and indeed do not think about—evaluation order also cannot predict which data structures will live for a long time. Since Haskell programs allocate objects very fast, if large numbers of them end up with long lifetimes, then the peak space requirements can be very high indeed. The next step was thus to extend the heap profiler to provide direct information about object lifetimes. This step was taken by Runciman and Röjemo (the author of `nhc`), who had by this time joined Runciman at the University of York. The new profiler could show how much of the heap contained data that was not yet needed (*lag*), would never be used again (*drag*), or, indeed, was never used at all (*void*) (Rjemo and Runciman, 1996a). A further extension introduced *retainer profiling*, which could explain *why* data was not garbage collected by showing which objects pointed at the data of interest (Rjemo and Runciman, 1996b). Combina-

tions of these forms made it possible for programmers to get answers to very specific questions about space use, such as “what kind of objects point at cons cells allocated by function `foo`, after their last use?” With information at this level of detail, Runciman and Röjemo were able to improve the peak space requirements of their `claify` program to less than 1K—three orders of magnitude better than the original version. They also achieved a factor-of-two improvement in the `nhc` compiler itself, which had already been optimised using their earlier tools.

10.3 Controlling evaluation order

In 1996, Haskell 1.3 introduced two features that give the programmer better control over evaluation order:

- the standard function `seq`, which evaluates its first argument, and then returns its second:

$$\text{seq } x \ y = \begin{cases} \perp, & \text{if } x = \perp \\ y, & \text{otherwise} \end{cases}$$

- strictness annotations in `data` definitions, as in:

```
data SList a = SNil | SCons !a !(SList a)
```

where the exclamation points denote strict fields, and thus here define a type of strict lists, whose elements are evaluated before the list is constructed.

Using these constructs, a programmer can move selected computations earlier, sometimes dramatically shortening the lifetimes of data structures. Both `seq` and strict components of data structures were already present in Miranda for the same reasons (Turner, 1985), and indeed `seq` had been used to fix space leaks in lazy programs since the early 1980s (Scheevel, 1984; Hughes, 1983).

Today, introducing a `seq` at a carefully chosen point is a very common way of fixing a space leak, but interestingly, this was not the main reason for introducing it into Haskell. On the contrary, `seq` was primarily introduced to improve the *speed* of Haskell programs! By 1996, we understood the importance of using strictness analysis to recognise strict functions, in order to invoke them using call-by-value rather than the more expensive call-by-need, but the results of strictness analysis were not always as good as we hoped. The reason was that many functions were “nearly,” but not quite, strict, and so the strictness analyser was forced to (safely) classify them as non-strict. By introducing calls of `seq`, the programmer could help the strictness analyser deliver better results. Strictness analysers were particularly poor at analysing data types, hence the introduction of strictness annotations in data type declarations, which not only made many more functions strict, but also allowed the compiler to optimise the representation of the data type in some cases.

Although `seq` was not introduced into Haskell primarily to fix space leaks, Hughes and Runciman were by this time well aware of its importance for this purpose. Runciman had spent a sabbatical at Chalmers in 1993, when he was working on his heap profiler and Hughes had a program with particularly stubborn space leaks—the two spent much time working together to track them down. This program was in LML, which already had `seq`, and time and again a carefully placed `seq` proved critical to plugging a leak. Hughes was very concerned that Haskell’s version of `seq` should support space debugging well.

But adding `seq` to Haskell was controversial because of its negative effect on semantic properties. In particular, `seq` is not definable in the lambda calculus, and is the only way to distinguish $\lambda x \rightarrow \perp$ from \perp (since `seq` $\perp 0$ goes into a loop, while `seq` $(\lambda x \rightarrow \perp) 0$ does not)—a distinction that Jon Fairbairn, in

particular, was dead set against making. Moreover, `seq` weakens the parametricity property that polymorphic functions enjoy, because `seq` does not satisfy the parametricity property for its type $\forall a,b. a \rightarrow b \rightarrow b$, and neither do polymorphic functions that use it. This would weaken Wadler’s “free theorems” in Haskell (Wadler, 1989) in a way that has recently been precisely characterised by Patricia Johann and Janis Voigtländer (Johann and Voigtländer, 2004).

Unfortunately, parametricity was by this time not just a nice bonus, but the justification for an important compiler optimisation, namely *deforestation*—the transformation of programs to eliminate intermediate data structures. Deforestation is an important optimisation for programs written in the “listful” style that Haskell encourages, but Wadler’s original transformation algorithm (Wadler, 1990b) had proven too expensive for daily use. Instead, GHC used *short-cut deforestation*, which depends on two combinators: `foldr`, which consumes a list, and

```
build g = g (: [])
```

which constructs one, with the property that

```
foldr k z (build g) = g k z
```

(the “`foldr/build rule`”) (Gill et al., 1993). Applying this rewrite rule from left to right eliminates an intermediate list very cheaply. It turns out that the `foldr/build` rule is not true for *any* function `g`; it holds only if `g` has a sufficiently polymorphic type, and that can in turn be guaranteed by giving `build` a rank-2 type (Section 6.7). The proof relies on the parametricity properties of `g`’s type.

This elegant use of parametricity to guarantee a sophisticated program transformation was cast into doubt by `seq`. Launchbury argued forcefully that parametricity was too important to give up, for this very reason. Hughes, on the other hand, was very concerned that `seq` should be applicable to values of *any* type—even type variables—so that space leaks could be fixed even in polymorphic code. These two goals are virtually incompatible. The solution adopted for Haskell 1.3 was to make `seq` an *overloaded* function, rather than a polymorphic one, thus weakening the parametricity property that it should satisfy. Haskell 1.3 introduced a class

```
class Eval a where
  strict :: (a->b) -> a -> b
  seq    :: a -> b -> b
  strict f x = x `seq` f x
```

with the suspect operations as its members. However, programmers were not allowed to define their own instances of this class—which might not have been strict (!)—instead its instances were derived automatically. The point of the `Eval` class was to record uses of `seq` in the *types* of polymorphic functions, as contexts of the form `Eval a =>`, thus warning the programmer and the compiler that parametricity properties in that type variable were restricted. Thus short-cut deforestation remained sound, while space leaks could be fixed at any type.

However, the limitations of this solution soon became apparent. Inspired by the Fox project at CMU, two of Hughes’s students implemented a TCP/IP stack in Haskell, making heavy use of polymorphism in the different layers. Their code turned out to contain serious space leaks, which they attempted to fix using `seq`. But whenever they inserted a call of `seq` on a type variable, the type signature of the enclosing function changed to require an `Eval` instance for that variable—just as the designers of Haskell 1.3 intended. But often, the type signatures of very many functions changed as a consequence of a single `seq`. This would not have mattered if the type signatures were inferred by the compiler—but the students had written them explicitly in their code. Moreover, they had done

so not from choice, but because Haskell’s monomorphism restriction *required* type signatures on these particular definitions (Section 6.2). As a result, each insertion of a `seq` became a nightmare, requiring repeated compilations to find affected type signatures and manual correction of each one. Since space debugging is to some extent a question of trial and error, the students needed to insert and remove calls of `seq` time and time again. In the end they were forced to conclude that fixing their space leaks was simply not feasible in the time available to complete the project—not because they were hard to find, but because making the necessary corrections was simply too heavyweight. This experience provided ammunition for the eventual removal of class `Eval` in Haskell 98.

Thus, today, `seq` is a simple polymorphic function that can be inserted or removed freely to fix space leaks, without changing the types of enclosing functions. We have sacrificed parametricity in the interests of programming agility and (sometimes dramatic) optimisations. GHC still uses short-cut deforestation, but it is unsound—for example, this equation does *not* hold

```
foldr ⊥ 0 (build seq) ≠ seq ⊥ 0
```

Haskell’s designers love semantics, but even semantics has its price. It’s worth noting that making programs stricter is not the only way to fix space leaks in Haskell. Object lifetimes can be shortened by moving their last use earlier—or by creating them later. In their famous case study, the first optimisation Runciman and Wakeling made was to make the program *more lazy*, delaying the construction of a long list until just before it was needed. Hearing Runciman describe the first heap profiler at a meeting of Working Group 2.8, Peter Lee decided to translate the code into ML to discover the effect of introducing strictness everywhere. Sure enough, his translation used only one third as much space as the lazy original—but Runciman and Wakeling’s first optimisation made the now-lazier program twice as efficient as Peter Lee’s version.

The extreme sensitivity of Haskell’s space use to evaluation order is a two-edged sword. Tiny changes—the addition or removal of a `seq` in one place—can dramatically change space requirements. On the one hand, it is very hard for programmers to *anticipate* their program’s space behaviour and place calls of `seq` correctly when the program is first written. On the other hand, given sufficiently good profiling information, space performance can be improved dramatically by very small changes in just the right place—*without* changing the overall structure of the program. As designers who believe in reasoning, we are a little ashamed that reasoning about space use in Haskell is so intractable. Yet Haskell encourages programmers—even forces them—to forget space optimisation until *after* the code is written, profiled, and the major space leaks found, and at that point puts powerful tools at the programmer’s disposal to fix them. Maybe this is nothing to be ashamed of, after all.

10.4 Debugging and tracing

Haskell’s rather unpredictable evaluation order also made conventional approaches to tracing and debugging difficult to apply. Most Haskell implementations provide a “function”

```
trace :: String -> a -> a
```

that prints its first argument as a side-effect, then returns its second—but it is not at all uncommon for the printing of the first argument to trigger *another* call of `trace` before the printing is complete, leading to very garbled output. To avoid such problems, more sophisticated debuggers aim to *abstract away* from the evaluation order.

10.4.1 Algorithmic debugging

One way to do so is via *algorithmic debugging* (Shapiro, 1983), an approach in which the debugger, rather than the user, takes the initiative to explore the program’s behaviour. The debugger presents function calls from a faulty run to the user, together with their arguments and results, and asks whether the result is correct. If not, the debugger proceeds to the calls made from the faulty one (its “children”), finally identifying a call with an incorrect result, all of whose children behaved correctly. This is then reported as the location of the bug.

Since algorithmic debugging just depends on the input-output behaviour of functions, it seems well suited to lazy programs. But there is a difficulty—the values of function arguments and (parts of their) results are often not computed until long *after* the function call is complete, because they are not needed until later. If they were computed early by an algorithmic debugger, in order to display them in questions to the user, then this itself might trigger faults or loops that would otherwise not have been a problem at all! Henrik Nilsson solved this problem in 1993 (Nilsson and Fritzson, 1994), in an algorithmic debugger for a small lazy language called Freja, by waiting until execution was complete before starting algorithmic debugging. At the end of program execution, it is known whether or not each value was required—if it was, then its value is now known and can be used in a question, and if it wasn’t, then the value was irrelevant to the bug anyway. This “post mortem” approach abstracts nicely from evaluation order, and has been used by all Haskell debuggers since.

Although Nilsson’s debugger did not handle Haskell, Jan Sparud was meanwhile developing one that did, by transforming Haskell program source code to collect debugging information while computing its result. Nilsson and Sparud then collaborated to combine and scale up their work, developing efficient methods to build “evaluation dependence trees” (Nilsson and Sparud, 1997), data structures that provided all the necessary information for post-mortem algorithmic debugging. Nilsson and Sparud’s tools are no longer extant, but the ideas are being pursued by Bernie Pope in his algorithmic debugger Buddha for Haskell 98 (Pope, 2005), and by the Hat tools described next.

10.4.2 Debugging via redex trails

In 1996, Sparud joined Colin Runciman’s group at the University of York to begin working on *redex trails*, another form of program trace which supports stepping backwards through the execution (Sparud and Runciman, 1997). Programmers can thus ask “*Why* did we call `f` with these arguments?” as well as inspect the evaluation of the call itself.

Runciman realised that, with a little generalisation, the *same* trace could be used to support several different kinds of debugging (Wallace et al., 2001). This was the origin of the new Hat project, which has developed a new tracer for Haskell 98 and a variety of trace browsing tools. Initially usable only with `nhc`, in 2002 Hat became a separate tool, working by source-to-source transformation, and usable with any Haskell 98 compiler. Today, there are trace browsers supporting redex-trail debugging, algorithmic debugging, observational debugging, single-stepping, and even test coverage measurement, together with several more specific tools for tracking down particular kinds of problem in the trace—see <http://www.haskell.org/hat/>. Since 2001, Runciman has regularly invited colleagues to send him their bugs, or even insert bugs into his own code while his back was turned, for the sheer joy of tracking them down with Hat!

The Hat suite are currently the most widely used debugging tools for Haskell, but despite their power and flexibility, they have not

become a regular part of programming for most users¹¹. This is probably because Haskell, as it is used in practice, has remained a moving target: new extensions appear frequently, and so it is hard for a language-aware tool such as Hat to keep up. Indeed, Hat was long restricted to Haskell 98 programs only—a subset to which few serious users restrict themselves. Furthermore, the key to Hat’s implementation is an ingenious, systematic source-to-source transformation of the entire program. This transformation includes the libraries (which are often large and use language extensions), and imposes a substantial performance penalty on the running program.

10.4.3 Observational debugging

A more lightweight idea was pursued by Andy Gill, who developed HOOD, the Haskell Object Observation Debugger, in 1999–2000 (Gill, 2000). HOOD is also a post-mortem debugger, but users indicate explicitly which information should be collected by inserting calls of

```
observe :: String -> a -> a
```

in the program to be debugged. In contrast to `trace`, `observe` prints nothing when it is called—it just collects the value of its second argument, tagged with the first. When execution is complete, all the collected values are printed, with values with the same tag gathered together. Thus the programmer can observe the collection of values that appeared at a program point, which is often enough to find bugs.

As in Nilsson and Sparud’s work, values that were collected but never evaluated are displayed as a dummy value “`_`”. For example,

```
Observe> take 2 (observe "nats" [0..])
[0,1]
>>>>> Observations <<<<<
nats
(0 : 1 : _)
```

This actually provides useful information about lazy evaluation, showing us *how much* of the input was needed to produce the given result.

HOOD can even observe function values, displaying them as a table of observed arguments and results—the same information that an algorithmic debugger would use to track down the bug location. However, HOOD leaves locating the bug to the programmer.

10.5 Testing tools

While debugging tools have not yet really reached the Haskell mainstream, testing tools have been more successful. The most widely used is QuickCheck, developed by Koen Claessen and Hughes. QuickCheck is based on a cool idea that turned out to work very well in practice, namely that programs can be tested against specifications by formulating specifications as boolean functions that should always return `True`, and then invoking these functions on random data. For example, the function definition

```
prop_reverse :: [Integer] -> [Integer] -> Bool
prop_reverse xs ys =
    reverse (xs++ys) == reverse ys++reverse xs
```

expresses a relationship between `reverse` and `++` that should always hold. The QuickCheck user can test that it does just by evaluating `quickCheck prop_reverse` in a Haskell interpreter. In this

¹¹In a web survey we conducted, only 3% of respondents named Hat as one of the “most useful tools and libraries.”

case testing succeeds, but when properties fail then QuickCheck displays a counter example. Thus, for the effort of writing a simple property, programmers can test a very large number of cases, and find counter examples very quickly.

To make this work for larger-than-toy examples, programmers need to be able to control the random generation. QuickCheck supports this via an abstract data type of “generators,” which conceptually represent sets of values (together with a probability distribution). For example, to test that insertion into an ordered list preserves ordering, the programmer could write

```
prop_insert :: Integer -> Bool
prop_insert x
  = forAll orderedList
    (\xs -> ordered (insert x xs))
```

We read the first line as quantification over the set of ordered lists, but in reality `orderedList` is a test data generator, which `forAll` invokes to generate a value for `xs`. QuickCheck provides a library of combinators to make such generators easy to define.

QuickCheck was first released in 1999 and was included in the GHC and Hugs distributions from July 2000, making it easily accessible to most users. A first paper appeared in 2000 (Claessen and Hughes, 2000), with a follow-up article on testing monadic code in 2002 (Claessen and Hughes, 2002). Some early success stories came from the annual ICFP programming contests: Tom Moertel (“Team Functional Beer”) wrote an account¹² of his entry in 2001, with quotable quotes such as “QuickCheck to the rescue!” and “Not so fast, QuickCheck spotted a corner case...,” concluding

QuickCheck found these problems and more, many that I wouldn’t have found without a massive investment in test cases, and it did so quickly and easily. From now on, I’m a QuickCheck man!

Today, QuickCheck is widely used in the Haskell community and is one of the tools that has been adopted by Haskell programmers in industry, even appearing in job ads from Galois Connections and Aetion Technologies. Perhaps QuickCheck has succeeded in part because of who Haskell programmers are: given the question “What is more fun, testing code or writing formal specifications?” many Haskell users would choose the latter—if you can test code by writing formal specifications, then so much the better!

QuickCheck is not only a useful tool, but also a good example of applying some of Haskell’s unique features. It defines a domain-specific language of testable properties, in the classic Haskell tradition. The class system is used to associate a test data generator with each type, and to overload the `quickCheck` function so that it can test properties with any number of arguments, of any types. The abstract data type of generators is a monad, and Haskell’s syntactic sugar for monads is exploited to make generators easy to write. The Haskell language thus had a profound influence on QuickCheck’s design.

This design has been emulated in many other languages. One of the most interesting examples is due to Christian Lindig, who found bugs in production-quality C compilers’ calling conventions by generating random C programs in a manner inspired by QuickCheck (Lindig, 2005). A port to Erlang has been used to find unexpected errors in a pre-release version of an Ericsson Media Gateway (Arts et al., 2006).

QuickCheck is not the only testing tool for Haskell. In 2002, Dean Herington released HUnit (Herington, 2002), a test framework inspired by the JUnit framework for Java, which has also acquired a

dedicated following. HUnit supports more traditional unit testing: it does not generate test cases, but rather provides ways to define test cases, structure them into a hierarchy, and run tests automatically with a summary of the results.

Part IV

Applications and Impact

A language does not have to have a direct impact on the real world to hold a prominent place in the history of programming languages. For example, Algol was never used substantially in the real world, but its impact was huge. On the other hand, impact on the real world was an important goal of the Haskell Committee, so it is worthwhile to consider how well we have achieved this goal.

The good news is that there are far too many interesting applications of Haskell to enumerate in this paper. The bad news is that Haskell is still not a mainstream language used by the masses! Nevertheless, there are certain niches where Haskell has fared well. In this section we discuss some of the more interesting applications and real-world impacts, with an emphasis on successes attributable to specific language characteristics.

11. Applications

Some of the most important applications of Haskell were originally developed as libraries. The Haskell standard includes a modest selection of libraries, but many more are available. The Haskell web site (haskell.org) lists more than a score of categories, with the average category itself containing a score of entries. For example, the Edison library of efficient data structures, originated by Okasaki (Okasaki, 1998a) and maintained by Robert Dockins, provides multiple implementations of sequences and collections, organised using type classes. The HSQL library interfaces to a variety of databases, including MySQL, Postgres, ODBC, SQLite, and Oracle; it is maintained by Angelov.

Haskell also has the usual complement of parser and lexer generators. Marlow’s *Happy* was designed to be similar to yacc and generated LALR parsers. (“Happy” is a “dyslexic acronym” for Yet Another Haskell Parser.) Paul Callaghan recently extended Happy to produce Generalised LR parsers, which work with ambiguous grammars, returning all possible parses. Parser combinator libraries are discussed later in this section. Documentation of Haskell programs is supported by several systems, including Marlow’s Hadock tool.

11.1 Combinator libraries

One of the earliest success stories of Haskell was the development of so-called *combinator libraries*. What is a combinator library? The reader will search in vain for a definition of this heavily used term, but the key idea is this: a combinator library offers functions (the combinators) that combine *functions* together to make bigger functions.

For example, an early paper that made the design of combinator libraries a central theme was Hughes’s paper “The design of a pretty-printing library” (Hughes, 1995). In this paper a “smart document” was an abstract type that can be thought of like this:

```
type Doc = Int -> String
```

That is, a document takes an `Int`, being the available width of the paper, and lays itself out in a suitable fashion, returning a `String`

¹² See <http://www.kuro5hin.org/story/2001/7/31/0102/11014>.

that can be printed. Now a library of combinators can be defined such as:

```
above   :: Doc -> Doc -> Doc
beside :: Doc -> Doc -> Doc
sep    :: [Doc] -> Doc
```

The function `sep` lays the subdocuments out beside each other if there is room, or above each other if not.

While a `Doc` can be *thought of* as a function, it may not be *implemented* as a function; indeed, this trade-off is a theme of Hughes's paper. Another productive way to think of a combinator library is as a *domain-specific language* (DSL) for describing values of a particular type (for example, document layout in the case of pretty-printing). DSLs in Haskell are described in more detail in Section 11.2.

11.1.1 Parser combinators

One of the most fertile applications for combinator libraries has undoubtedly been *parser combinators*. Like many ingenious programming techniques, this one goes back to Burge's astonishing book *Recursive Programming Techniques* (Burge, 1975), but it was probably Wadler's paper "How to replace failure by a list of successes" (Wadler, 1985) that brought it wider attention, although he did not use the word "combinator" and described the work as "folklore".

A parser may be thought of as a function:

```
type Parser = String -> [String]
```

That is, a `Parser` takes a string and attempts to parse it, returning zero or more depleted input strings, depending on how many ways the parse could succeed. Failure is represented by the empty list of results. Now it is easy to define a library of combinators that combine parsers together to make bigger parsers, and doing so allows an extraordinarily direct transcription of BNF into executable code. For example, the BNF

```
float ::= sign? digit+ ('.' digit+)?
```

might translate to this Haskell code:

```
float :: Parser
float = optional sign <*> oneOrMore digit <*>
       optional (lit '.' <*> oneOrMore digit)
```

The combinators `optional`, `oneOrMore`, and `(<*>)` combine parsers to make bigger parsers:

```
optional, oneOrMore :: Parser -> Parser
(<*>) :: Parser -> Parser -> Parser
```

It is easy for the programmer to make new parser combinators by combining existing ones.

A parser of this kind is only a *recogniser* that succeeds or fails. Usually, however, one wants a parser to return a value as well, a requirement that dovetails precisely with Haskell's notion of a monad (Section 7). The type of parsers is parameterised to `Parser t`, where `t` is the type of value returned by the parser. Now we can write the `float` parser using `do`-notation, like this:

```
float :: Parser Float
float
= do mb_sgn  <- optional sign
     digs    <- oneOrMore digit
     mb_frac <- optional (do lit '.'
                           oneOrMore digit)
     return (mkFloat mb_sgn digs mb_frac)
```

where `optional :: Parser a -> Parser (Maybe a)`, and `oneOrMore :: Parser a -> Parser [a]`.

The interested reader may find the short tutorial by Hutton and Meijer helpful (Hutton and Meijer, 1998). There are dozens of papers about cunning variants of parser combinators, including error-correcting parsers (Swierstra and Duponcheel, 1996), parallel parsing (Claessen, 2004), parsing permutation phrases (Baars et al., 2004), packrat parsing (Ford, 2002), and lexical analysis (Chakravarty, 1999b). In practice, the most complete and widely used library is probably Parsec, written by Daan Leijen.

11.1.2 Other combinator libraries

In a way, combinator libraries do not embody anything fundamentally new. Nevertheless, the idea has been extremely influential, with dozens of combinator libraries appearing in widely different areas. Examples include pretty printing (Hughes, 1995; Wadler, 2003), generic programming (Lämmel and Peyton Jones, 2003), embedding Prolog in Haskell (Spivey and Seres, 2003), financial contracts (Peyton Jones et al., 2000), XML processing (Wallace and Runciman, 1999), synchronous programming (Scholz, 1998), database queries (Leijen and Meijer, 1999), and many others.

What makes Haskell such a natural fit for combinator libraries? Aside from higher-order functions and data abstraction, there seem to be two main factors, both concerning laziness. First, one can write recursive combinators without fuss, such as this recursive parser for terms:

```
term :: Parser Term
term = choice [ float, integer,
                variable, parens term, ... ]
```

In call-by-value languages, recursive definitions like this are generally not allowed. Instead, one would have to eta-expand the definition, thereby cluttering the code and (much more importantly) wrecking the abstraction (Syme, 2005).

Second, laziness makes it extremely easy to write combinator libraries with unusual control flow. Even in Wadler's original list-of-successes paper, laziness plays a central role, and that is true of many other libraries mentioned above, such as embedding Prolog and parallel parsing.

11.2 Domain-specific embedded languages

A common theme among many successful Haskell applications is the idea of writing a library that turns Haskell into a *domain-specific embedded language* (DSEL), a term first coined by Hudak (Hudak, 1996a; Hudak, 1998). Such DSELs have appeared in a diverse set of application areas, including graphics, animation, vision, control, GUIs, scripting, music, XML processing, robotics, hardware design, and more.

By "embedded language" we mean that the domain-specific language is simply an extension of Haskell itself, sharing its syntax, function definition mechanism, type system, modules and so on. The "domain-specific" part is just the new data types and functions offered by a library. The phrase "embedded language" is commonly used in the Lisp community, where Lisp macros are used to design "new" languages; in Haskell, thanks to lazy evaluation, much (although emphatically not all) of the power of macros is available through ordinary function definitions. Typically, a data type is defined whose essential nature is often, at least conceptually, a function, and operators are defined that combine these abstract functions into larger ones of the same kind. The final program is then "executed" by decomposing these larger pieces and applying the embedded functions in a suitable manner.

In contrast, a non-embedded DSL can be implemented by writing a conventional parser, type checker, and interpreter (or compiler) for the language. Haskell is very well suited to such ap-

proaches as well. However, Haskell has been particularly successful for domain-specific embedded languages. Below is a collection of examples.

11.2.1 Functional Reactive Programming

In the early 1990s, Conal Elliott, then working at Sun Microsystems, developed a DSL called *TBAG* for constraint-based, semi-declarative modelling of 3D animations (Elliott et al., 1994; Schechter et al., 1994). Although largely declarative, *TBAG* was implemented entirely in C++. The success of his work resulted in Microsoft hiring Elliott and a few of his colleagues into the graphics group at Microsoft Research. Once at Microsoft, Elliott's group released in 1995 a DSL called *ActiveVRML* that was more declarative than *TBAG*, and was in fact based on an ML-like syntax (Elliott, 1996). It was about that time that Elliott also became interested in Haskell, and began collaborating with several people in the Haskell community on implementing ActiveVRML in Haskell. Collaborations with Hudak at Yale on design issues, formal semantics, and implementation techniques led in 1998 to a language that they called *Fran*, which stood for “functional reactive animation” (Elliott and Hudak, 1997; Elliott, 1997).

The key idea in *Fran* is the notion of a *behaviour*, a first-class data type that represents a *time-varying* value. For example, consider this *Fran* expression:

```
pulse :: Behavior Image
pulse = circle (sin time)
```

In *Fran*, *pulse* is a time-varying image value, describing a circle whose radius is the sine of the time, in seconds, since the program began executing. A good way to understand behaviours is via the following data type definition:

```
newtype Behavior a = Beh (Time -> a)
type Time = Float
```

That is, a behaviour in *Fran* is really just a function from time to values. Using this representation, the value *time* used in the *pulse* example would be defined as:

```
time :: Behaviour Time
time = Beh (\t -> t)
```

i.e., the identity function. Since many *Fran* behaviours are numeric, Haskell's *Num* and *Floating* classes (for example) allow one to specify how to add two behaviours or take the sine of a behaviour, respectively:

```
instance Num (Behavior a) where
  Beh f + Beh g = Beh (\t -> f t + g t)

instance Floating (Behaviour a) where
  sin (Beh f) = Beh (\t -> sin (f t))
```

Thinking of behaviours as functions is perhaps the easiest way to reason about *Fran* programs, but of course behaviours are abstract, and thus can be implemented in other ways, just as with combinator libraries described earlier.

Another key idea in *Fran* is the notion of an infinite stream of *events*. Various “switching” combinators provide the connection between behaviours and events—i.e. between the continuous and the discrete—thus making *Fran*-like languages suitable for so-called “hybrid systems.”

This work, a classic DSEL, was extremely influential. In particular, Hudak's research group and others began a flurry of research strands which they collectively referred to as *functional reactive programming*, or FRP. These efforts included: the application of

FRP to real-world physical systems, including both mobile and humanoid robots (Peterson et al., 1999a; Peterson et al., 1999b); the formal semantics of FRP, both denotational and operational, and the connection between them (Wan and Hudak, 2000); real-time variants of FRP targeted for real-time embedded systems (Wan et al., 2002; Wan et al., 2001; Wan, 2002); the development of an arrow-based version of FRP called *Yampa* in 2002, that improves both the modularity and performance of previous implementations (Hudak et al., 2003); the use of FRP and Yampa in the design of graphical user interfaces (Courtney and Elliott, 2001; Courtney, 2004; Sage, 2000) (discussed further in Section 11.3); and the use of Yampa in the design of a 3D first-person shooter game called *Frag* in 2005 (Cheong, 2005). Researchers at Brown have more recently ported the basic ideas of FRP into a Scheme environment called “Father Time” (Cooper and Krishnamurthi, 2006).

11.2.2 XML and web-scripting languages

Demonstrating the ease with which Haskell can support domain-specific languages, Wallace and Runciman were one of the first to extend an existing programming language with features for XML programming, with a library and toolset called HaXml (Wallace and Runciman, 1999). They actually provided two approaches to XML processing. One was a small combinator library for manipulating XML, that captured in a uniform way much of the same functionality provided by the XPath language at the core of XSLT (and later XQuery). The other was a data-binding approach (implemented as a pre-processor) that mapped XML data onto Haskell data structures, and vice versa. The two approaches have complementary strengths: the combinator library is flexible but all XML data has the same type; the data-binding approach captures more precise types but is less flexible. Both approaches are still common in many other languages that process XML, and most of these languages still face the same trade-offs.

Haskell was also one of the first languages to support what has become one of the standard approaches to implementing web applications. The traditional approach to implementing a web application requires breaking the logic into one separate program for each interaction between the client and the web server. Each program writes an HTML form, and the responses to this form become the input to the next program in the series. Arguably, it is better to invert this view, and instead to write a single program containing calls to a primitive that takes an HTML form as argument and returns the responses as the result, and this approach was first taken by the domain-specific language MAWL (Atkins et al., 1999).

However, one does not need to invent a completely new language for the purpose; instead, this idea can be supported using concepts available in functional languages, either continuations or monads (the two approaches are quite similar). Paul Graham used a continuation-based approach as the basis for one of the first commercial applications for building web stores, which later became Yahoo Stores (Graham, 2004). The same approach was independently discovered by Christian Queinnec (Queinnec, 2000) and further developed by Matthias Felleisen and others in PLT Scheme (Graunke et al., 2001). Independently, an approach based on a generalisation of monads called arrows was discovered by Hughes (Hughes, 2000) (Section 6.7). Hughes's approach was further developed by Peter Thiemann in the WASH system for Haskell, who revised it to use monads in place of arrows (Thiemann, 2002b). It turns out that the approach using arrows or monads is closely related to the continuation approach (since continuations arise as a special case of monads or arrows). The continuation approach has since been adopted in a number of web frameworks widely used by developers, such as Seaside and RIFE.

Most of this work has been done in languages (Scheme, Smalltalk, Ruby) without static typing. Thiemann's work has shown that the same approach works with a static type system that can guarantee that the type of information returned by the form matches the type of information that the application expects. Thiemann also introduced a sophisticated use of type classes to ensure that HTML or XML used in such applications satisfies the regular expression types imposed by the document type declarations (DTD) used in XML (Thiemann, 2002a).

11.2.3 Hardware design languages

Lazy functional languages have a long history of use for describing and modelling synchronous hardware, for two fundamental reasons: first, because lazy streams provide a natural model for discrete time-varying signals, making simulation of functional models very easy; and second, because higher-order functions are ideal for expressing the regular structure of many circuits. Using lazy streams dates to Steve Johnson's work in the early eighties, for which he won the ACM Distinguished Dissertation award in 1984 (Johnson, 1984). Higher-order functions for capturing regular circuit structure were pioneered by Mary Sheeran in her language μ FP (Sheeran, 1983; Sheeran, 1984), inspired by Backus' FP (Backus, 1978b).

It was not long before Haskell too was applied to this domain. One of the first to do so was John O'Donnell, whose Hydra hardware description language is embedded in Haskell (O'Donnell, 1995). Another was Dave Barton at Intermetrics, who proposed MHDL (Microwave Hardware Description Language) based on Haskell 1.2 (Barton, 1995). This was one of the earliest signs of industrial interest in Haskell, and Dave Barton was later invited to join the Haskell Committee as a result.

A little later, Launchbury and his group used Haskell to describe microprocessor architectures in the Hawk system (Matthews et al., 1998), and Mary Sheeran et al. developed Lava (Bjesse et al., 1998), a system for describing regular circuits in particular, which can simulate, verify, and generate net-lists for the circuits described. Both Hawk and Lava are examples of domain-specific languages embedded in Haskell.

When Satnam Singh moved to Xilinx in California, he took Lava with him and added the ability to generate FPGA layouts for Xilinx chips from Lava descriptions. This was one of the first successful industrial applications of Haskell: Singh was able to generate highly efficient and reconfigurable cores for accelerating applications such as Adobe Photoshop (Singh and Slous, 1998). For years thereafter, Singh used Lava to develop specialised core generators, delivered to Xilinx customers as compiled programs that, given appropriate parameters, generated important parts of an FPGA design—in most cases without anyone outside Xilinx being aware that Haskell was involved! Singh tells an amusing anecdote from these years: on one occasion, a bug in GHC prevented his latest core generator from compiling. Singh mailed his code to Peyton Jones at Microsoft Research, who was able to compile it with the development version of GHC, and sent the result back to Singh the next day. When Singh told his manager, the manager exclaimed incredulously, “You mean to say you got 24-hour support from Microsoft?”

Lava in particular exercised Haskell's ability to embed domain specific languages to the limit. Clever use of the class system enables signals-of-lists and lists-of-signals, for example, to be used almost interchangeably, without a profusion of zips and unzips. Capturing sharing proved to be particularly tricky, though. Consider the following code fragment:

```
let x = nand a b
    y = nand a b
in ...
```

Here it seems clear that the designer intends to model two separate NAND-gates. But what about

```
let x = nand a b
    y = x
in ...
```

Now, clearly, the designer intends to model a single NAND-gate whose output signal is shared by x and y . Net-lists generated from these two descriptions should therefore be *different*—yet according to Haskell's intended semantics, these two fragments should be indistinguishable. For a while, Lava used a “circuit monad” to make the difference observable:

```
do x <- nand a b
    y <- nand a b
    ...
    ...
```

versus

```
do x <- nand a b
    y <- return x
    ...
    ...
```

which are perfectly distinguishable in Haskell. This is the recommended “Haskellish” approach—yet adopting a monadic syntax uniformly imposes quite a heavy cost on Lava users, which is frustrating given that the only reason for the monad is to distinguish sharing from duplication! Lava has been used to teach VLSI design to electrical engineering students, and in the end, the struggle to teach monadic Lava syntax to non-Haskell users became too much. Claessen used `unsafePerformIO` to implement “observable sharing”, allowing Lava to use the first syntax above, but still to distinguish sharing from duplication when generating net-lists, theorem-prover input, and so on. Despite its unsafe implementation, observable sharing turns out to have a rather tractable theory (Claessen and Sands, 1999), and thus Lava has both tested Haskell's ability to embed other languages to the limit, and contributed a new mechanism to extend its power.

Via this and other work, lazy functional programming has had an important impact on industrial hardware design. Intel's large-scale formal verification work is based on a lazy language, in both the earlier Forté and current IDV systems. Sandburst was founded by Arvind to exploit Bluespec, a proprietary hardware description language closely based on Haskell (see Section 12.4.2). The language is now being marketed (with a System Verilog front end) by a spin-off company called Bluespec, but the tools are still implemented in Haskell.

A retrospective on the development of the field, and Lava in particular, can be found in Sheeran's JUCS paper (Sheeran, 2005).

11.2.4 Computer music

Haskore is a computer music library written in Haskell that allows expressing high-level musical concepts in a purely declarative way (Hudak et al., 1996; Hudak, 1996b; Hudak, 2003). Primitive values corresponding to notes and rests are combined using combinators for sequential and parallel composition to form larger musical values. In addition, musical ornamentation and embellishment (legato, crescendo, etc.) are treated by an object-oriented approach to musical instruments to provide flexible degrees of interpretation.

The first version of Haskore was written in the mid '90s by Hudak and his students at Yale. Over the years it has matured in a number of different ways, and aside from the standard distribution at Yale,

Henning Thielemann maintains an open-source Darcs repository (Section 12.3) to support further development. Haskore has been used as the basis of a number of computer music projects, and is actively used for computer music composition and education. One of the more recent additions to the system is the ability to specify musical sounds—i.e. instruments—in a declarative way, in which oscillators, filters, envelope generators, etc. are combined in a signal-processing-like manner.

Haskore is based on a very simple declarative model of music with nice algebraic properties that can, in fact, be generalized to other forms of time-varying media (Hudak, 2004). Although many other computer music languages preceded Haskore, none of them, perhaps surprisingly, reflects this simple structure. Haskell’s purity, lazy evaluation, and higher-order functions are the key features that make possible this elegant design.

11.2.5 Summary

Why has Haskell been so successful in the DSEL arena? After all, many languages provide the ability to define new data types together with operations over them, and a DSEL is little more than that! No single feature seems dominant, but we may identify the following ways in which Haskell is a particularly friendly host language for a DSEL:

1. *Type classes* permit overloading of many standard operations (such as those for arithmetic) on many nonstandard types (such as the `Behaviour` type above).
2. *Higher-order functions* allow encoding nonstandard behaviours and also provide the glue to combine operations.
3. *Infix syntax* allows one to emulate infix operators that are common in other domains.
4. *Over-loaded numeric literals* allow one to use numbers in new domains without tagging or coercing them in awkward ways.
5. *Monads* and *arrows* are flexible mechanisms for combining operations in ways that reflect the semantics of the intended domain.
6. *Lazy evaluation* allows writing recursive definitions in the new language that are well defined in the DSEL, but would not terminate in a strict language.

The reader will also note that there is not much difference in concept between the combinator libraries described earlier and DSELS. For example, a parser combinator library can be viewed as a DSEL for BNF, which is just a meta-language for context-free grammars. And Haskell libraries for XML processing share a lot in common with parsing and layout, and thus with combinator libraries. It is probably only for historical reasons that one project might use the term “combinator library” and another the term “DSL” (or “DSEL”).

11.3 Graphical user interfaces

Once Haskell had a sensible I/O system (Section 7), the next obvious question was how to drive a graphical user interface (GUI). People interested in this area rapidly split into two groups: the *idealists* and the *pragmatists*.

The idealists took a radical approach. Rather than adopt the imperative, event-loop-based interaction model of mainstream programming languages, they sought to answer the question, “What is the right way to interact with a GUI in a purely declarative setting?” This question led to several quite unusual GUI systems:

- The *Fudgets* system was developed by Magnus Carlsson and Thomas Hallgren, at Chalmers University in Sweden. They treated the GUI as a network of “*stream processors*”, or stream

transformers (Carlsson and Hallgren, 1993). Each processor had a visual appearance, as well as being connected to other stream processors, and the shape of the network could change dynamically. There was no central event loop: instead each stream processor processed its own individual stream of events.

- Sigbjorn Finne, then a research student at Glasgow, developed *Haggis*, which replaced the event loop with extremely lightweight concurrency; for example, each button might have a thread dedicated to listening for clicks on that button. The stress was on widget *composition*, so that complex widgets could be made by composing together simpler ones (Finne and Peyton Jones, 1995). The requirements of Haggis directly drove the development of Concurrent Haskell (Peyton Jones et al., 1996).
- Based on ideas in Fran (see section 11.2.1), Meurig Sage developed *FranTk* (Sage, 2000), which combined the best ideas in Fran with those of the GUI toolkit Tk, including an imperative model of call-backs.
- Antony Courtney took a more declarative approach based entirely on FRP and Yampa, but with many similarities to Fudgets, in a system that he called *Fruit* (Courtney and Elliott, 2001; Courtney, 2004). Fruit is purely declarative, and uses arrows to “wire together” GUI components in a data-flow-like style.

Despite the elegance and innovative nature of these GUIs, none of them broke through to become the GUI toolkit of choice for a critical mass of Haskell programmers, and they all remained single-site implementations with a handful of users. It is easy to see why. First, developing a fully featured GUI is a huge task, and each system lacked the full range of widgets, and snazzy appearance, that programmers have come to expect. Second, the quest for purity always led to programming inconvenience in one form or another. The search for an elegant, usable, declarative GUI toolkit remains open.

Meanwhile, the pragmatists were not idle. They just wanted to get the job done, by the direct route of interfacing to some widely available GUI toolkit library, a so-called “binding.” Early efforts included an interface to Tcl/Tk called swish (Sinclair, 1992), and an interface to X windows (the Yale Haskell project), but there were many subsequent variants (e.g., TkGofer, TclHaskell, HTK) and bindings to other tool kits such as OpenGL (HOpenGL), GTK (e.g., Gtk2Hs, Gtk+Hs) and WxWidgets (WxHaskell). These efforts were hampered by the absence of a well defined foreign-function interface for Haskell, especially as the libraries involved have huge interfaces. As a direct result, early bindings were often somewhat compiler specific, and implemented only part of the full interface. More recent bindings, such as Gtk2Hs and WxHaskell, are generated automatically by transforming the machine-readable descriptions of the library API into the Haskell 98 standard FFI.

These bindings all necessarily adopt the interaction model of the underlying toolkit, invariably based on imperative widget creation and modification, together with an event loop and call-backs. Nevertheless, their authors often developed quite sophisticated Haskell wrapper libraries that present a somewhat higher-level interface to the programmer. A notable example is the Clean graphical I/O library, which formed an integral part of the Clean system from a very early stage (Achten et al., 1992) (unlike the fragmented approach to GUIs taken by Haskell). The underlying GUI toolkit for Clean was the Macintosh, but Clean allows the user to specify the interface by means of a data structure containing call-back functions. Much later, the Clean I/O library was ported to Haskell (Achten and Peyton Jones, 2000).

To this day, the Haskell community periodically agonises over the absence of a single standard Haskell GUI. Lacking such a standard is undoubtedly an inhibiting factor on Haskell’s development. Yet no one approach has garnered enough support to become *the* design, despite various putative standardisation efforts, although Wx-Haskell (another side project of the indefatigable Daan Leijen) has perhaps captured the majority of the pragmatist market.

11.4 Operating Systems

An early operating system for Haskell was hOp, a micro-kernel based on the runtime system of GHC, implemented by Sebastian Carlier and Jeremy Bobbio (Carlier and Bobbio, 2004). Building on hOp, a later project, House, implemented a system in which the kernel, window system, and all device drivers are written in Haskell (Hallgren et al., 2005). It uses a monad to provide access to the Intel IA32 architecture, including virtual memory management, protected execution of user binaries, and low-level IO operations.

11.5 Natural language processing¹³

Haskell has been used successfully in the development of a variety of natural language processing systems and tools. Richard Frost (Frost, 2006) gives a comprehensive review of relevant work in Haskell and related languages, and discusses new tools and libraries that are emerging, written in Haskell and related languages. We highlight two substantial applications that make significant use of Haskell.

Durham’s *LOLITA* system (*Large-scale, Object-based, Linguistic Interactor, Translator and Analyzer*) was developed by Garigliano and colleagues at the University of Durham (UK) between 1986 and 2000. It was designed as a general-purpose tool for processing unrestricted text that could be the basis of a wide variety of applications. At its core was a semantic network containing some 90,000 interlinked concepts. Text could be parsed and analysed then incorporated into the semantic net, where it could be reasoned about (Long and Garigliano, 1993). Fragments of semantic net could also be rendered back to English or Spanish. Several applications were built using the system, including financial information analysers and information extraction tools for Darpa’s “Message Understanding Conference Competitions” (MUC-6 and MUC-7). The latter involved processing original Wall Street Journal articles, to perform tasks such as identifying key job changes in businesses and summarising articles. LOLITA was one of a small number of systems worldwide to compete in all sections of the tasks. A system description and an analysis of the MUC-6 results were written by Callaghan (Callaghan, 1998).

LOLITA was an early example of a substantial application written in a functional language: it consisted of around 50,000 lines of Haskell (with around 6000 lines of C). It is also a complex and demanding application, in which many aspects of Haskell were invaluable in development. LOLITA was designed to handle unrestricted text, so that ambiguity at various levels was unavoidable and significant. Laziness was essential in handling the explosion of syntactic ambiguity resulting from a large grammar, and it was much used with semantic ambiguity too. The system used multiple DSELs (Section 11.2) for semantic and pragmatic processing and for generation of natural language text from the semantic net. Also important was the ability to work with complex abstractions and to prototype new analysis algorithms quickly.

The *Grammatical Framework* (GF) (Ranta, 2004) is a language for defining grammars based on type theory, developed by Ranta and colleagues at Chalmers University. GF allows users to describe a

precise abstract syntax together with one or more concrete syntaxes; the same description specifies both how to parse concrete syntax into abstract syntax, and how to linearise the abstract syntax into concrete syntax. An editing mode allows incremental construction of well formed texts, even using multiple languages simultaneously. The GF system has many applications, including high-quality translation, multi-lingual authoring, verifying mathematical proof texts and software specifications, communication in controlled language, and interactive dialogue systems. Many reusable “resource grammars” are available, easing the construction of new applications.

The main GF system is written in Haskell and the whole system is open-source software (under a GPL licence). Haskell was chosen as a suitable language for this kind of system, particularly for the compilation and partial evaluation aspects (of grammars). Monads and type classes are extensively used in the implementation.

12. The impact of Haskell

Haskell has been used in education, by the open-source community, and by companies. The language is the focal point of an active and still-growing user community. In this section we survey some of these groups of users and briefly assess Haskell’s impact on other programming languages.

12.1 Education

One of the explicit goals of Haskell’s designers was to create a language suitable for teaching. Indeed, almost as soon as the language was defined, it was being taught to undergraduates at Oxford and Yale, but initially there was a dearth both of textbooks and of robust implementations suitable for teaching. Both problems were soon addressed. The first Haskell book—Tony Davie’s *An Introduction to Functional Programming Systems Using Haskell*—appeared in 1992. The release of Gofer in 1991 made an “almost Haskell” system available with a fast, interactive interface, good for teaching. In 1995, when Hugs was released, Haskell finally had an implementation perfect for teaching—which students could also install and use on their PCs at home. In 1996, Simon Thompson published a Haskell version of his *Craft of Functional Programming* textbook, which had first appeared as a Miranda textbook a year earlier. This book (revised in 1998) has become the top-selling book on Haskell, far ahead of its closest competitor in Amazon’s sales rankings.

The arrival of Haskell 98 gave textbooks another boost. Bird revised *Introduction to Functional Programming*, using Haskell, in 1998, and in the same year Okasaki published the first textbook to use Haskell to teach another subject—*Purely Functional Data Structures*. This was followed the next year by Fethi Rabhi and Guy Lapalme’s algorithms text *Algorithms: A functional programming approach*, and new texts continue to appear, such as Graham Hutton’s 2006 book *Programming in Haskell*.

The first Haskell texts were quite introductory in nature, intended for teaching functional programming to first-year students. At the turn of the millennium, textbooks teaching more advanced techniques began to appear. Hudak’s *Haskell School of Expression* (Hudak, 2000) uses multimedia applications (such as graphics, animation, and music) to teach Haskell idioms in novel ways that go well beyond earlier books. A unique aspect of this book is its use of DSELs (for animation, music, and robotics) as an underlying theme (see Section 11.2). Although often suggested for first-year teaching, it is widely regarded as being more suitable for an advanced course. In 2002, Gibbons and de Moor edited *The Fun of Programming*, an advanced book on Haskell programming with contributions by many authors, dedicated to Richard Bird and intended as a follow-up to his text.

¹³This section is based on material contributed by Paul Callaghan.

Another trend is to teach discrete mathematics and logic using Haskell as a medium of instruction, exploiting Haskell's mathematical look and feel. Cordelia Hall and John O'Donnell published the first textbook taking this approach in 2000—*Discrete Mathematics Using a Computer*. Rex Page carried out a careful three-year study, in which students were randomly assigned to a group taught discrete mathematics in the conventional way, or a group taught using Hall and O'Donnell's text, and found that students in the latter group became significantly more effective programmers (Page, 2003). Recently (in 2004) Doets and van Eijck have published another textbook in this vein, *The Haskell Road to Logic, Maths and Programming*, which has rapidly become popular.

For the more advanced students, there has been an excellent series of International Summer Schools on Advanced Functional Programming, at which projects involving Haskell have always had a significant presence. There have been five such summer schools to date, held in 1995, 1996, 1998, 2002, and 2004.

12.1.1 A survey of Haskell in higher education

To try to form an impression of the use of Haskell in university education today, we carried out a web survey of courses taught in the 2005–2006 academic year. We make no claim that our survey is complete, but it was quite extensive: 126 teachers responded, from 89 universities in 22 countries; together they teach Haskell to 5,000–10,000 students every year¹⁴. 25% of these courses began using Haskell only in the last two years (since 2004), which suggests that the use of Haskell in teaching is currently seeing rapid growth.

Enthusiasts have long argued that functional languages are ideally suited to teaching introductory programming, and indeed, most textbooks on Haskell programming are intended for that purpose. Surprisingly, only 28 of the courses in our survey were aimed at beginners (i.e. taught in the first year, or assuming no previous programming experience). We also asked respondents which programming languages students learn first and second at their Universities, on the assumption that basic programming will teach at least two languages. We found that—even at Universities that teach Haskell—Java was the first language taught in 47% of cases, and also the most commonly taught second language (in 22% of cases). Haskell was among the first two programming languages only in 35% of cases (15% as first language, 20% as second language). However, beginners' courses did account for the largest single group of students to study Haskell, 2–4,000 every year, because each such course is taken by more students on average than later courses are.

The most common courses taught using Haskell are explicitly intended to teach functional programming *per se* (or sometimes declarative programming). We received responses from 48 courses of this type, with total student numbers of 1,300–2,900 per year. A typical comment from respondents was that the course was intended to teach “a different style of programming” from the object-oriented paradigm that otherwise predominates. Four other more advanced programming courses (with 3–700 students) can be said to have a similar aim.

The third large group of courses we found were programming language courses—ranging from comparative programming languages through formal semantics. There were 25 such courses, with 800–1,700 students annually. Surprisingly, there is currently no Haskell-based textbook aimed at this market—an opportunity, perhaps?

¹⁴ We asked only for approximate student numbers, hence the wide range of possibilities.

Haskell is used to teach nine compilers courses, with 3–700 students. It is also used to teach six courses in theoretical computer science (2–400 students). Both take advantage of well-known strengths of the language—symbolic computation and its mathematical flavour. Finally, there are two courses in hardware description (50–100 students), and one course in each of domain-specific languages, computer music, quantum computing, and distributed and parallel programming—revealing a surprising variety in the subjects where Haskell appears.

Most Haskell courses are aimed at experienced programmers seeing the language for the first time: 85% of respondents taught students with prior programming experience, but only 23% taught students who already knew Haskell. The years in which Haskell courses are taught are shown in this table:

Year	% ge
1st undergrad	20%
2nd undergrad	23%
3rd undergrad	25%
4–5th undergrad	16%
Postgrad	12%

This illustrates once again that the majority of courses are taught at more advanced levels.

The countries from which we received most responses were the USA (22%), the UK (19%), Germany (11%), Sweden (8%), Australia (7%), and Portugal (5%).

How does Haskell measure up in teaching? Some observations we received were:

- Both respondents and their students are generally happy with the choice of language—“Even though I am not a FL researcher, I enjoy teaching the course more than most of my other courses and students also seem to like the course.”
- Haskell attracts good students—“The students who take the Haskell track are invariably among the best computer science students I have taught.”
- Fundamental concepts such as types and recursion are hammered home early.
- Students can tackle more ambitious and interesting problems earlier than they could using a language like Java.
- Simple loop programs can be harder for students to grasp when expressed using recursion.
- The class system causes minor irritations, sometimes leading to puzzling error messages for students.
- Array processing and algorithms using in-place update are messier in Haskell.
- Haskell input/output is not well covered by current textbooks: “my impression was that students are mostly interested in things which Simon Peyton Jones addressed in his paper ‘Tackling the Awkward Squad’ (Peyton Jones, 2001). I think, for the purpose of teaching FP, we are in dire need of a book on FP that not only presents the purely functional aspects, but also comprehensively covers issues discussed in that paper.”

As mentioned earlier, a simplified version of Haskell, *Helium*, is being developed at Utrecht specifically for teaching—the first release was in 2002. Helium lacks classes, which enables it to give clearer error messages, but then it also lacks textbooks and the ability to “tackle the awkward squad.” It remains to be seen how successful it will be.

12.2 Haskell and software productivity

Occasionally we hear anecdotes about Haskell providing an “order-of-magnitude” reduction in code size, program development time, software maintenance costs, or whatever. However, it is very difficult to conduct a rigorous study to substantiate such claims, for any language.

One attempt at such a study was an exercise sponsored by Darpa (the U.S. Defense Advanced Research Projects Agency) in the early 1990s. About ten years earlier, Darpa had christened Ada as the standard programming language to be used for future software development contracts with the U.S. government. Riding on that wave of wisdom, they then commissioned a program called *ProtoTech* to develop software prototyping technology, including the development of a “common prototyping language,” to help in the design phase of large software systems. Potential problems associated with standardisation efforts notwithstanding, Darpa’s ProtoTech program funded lots of interesting programming language research, including Hudak’s effort at Yale.

Toward the end of the ProtoTech Program, the Naval Surface Warfare Center (NSWC) conducted an experiment to see which of many languages—some new (such as Haskell) and some old (such as Ada and C++)—could best be used to prototype a “geometric region server.” Ten different programmers, using nine different programming languages, built prototypes for this software component. Mark Jones, then a Research Scientist at Yale, was the primary Haskell programmer in the experiment. The results, described in (Carlson et al., 1993), although informal and partly subjective and too lengthy to describe in detail here, indicate fairly convincingly the superiority of Haskell in this particular experiment.

Sadly, nothing of substance ever came from this experiment. No recommendations were made to use Haskell in any kind of government software development, not even in the context of prototyping, an area where Haskell could have had significant impact. The community was simply not ready to adopt such a radical programming language.

In recent years there have been a few other informal efforts at running experiments of this sort. Most notably, the functional programming community, through ICFP, developed its very own Programming Contest, a three-day programming sprint that has been held every year since 1998. These contests have been open to anyone, and it is common to receive entries written in C and other imperative languages, in addition to pretty much every functional language in common use. The first ICFP Programming Contest, run by Olin Shivers in 1998, attracted 48 entries. The contest has grown substantially since then, with a peak of 230 entries in 2004—more teams (let alone team members) than conference participants! In every year only a minority of the entries are in functional languages; for example in 2004, of the 230 entries, only 67 were functional (24 OCaml, 20 Haskell, 12 Lisp, 9 Scheme, 2 SML, 1 Mercury, 1 Erlang). Nevertheless, functional languages dominate the winners: of the first prizes awarded in the eight years of the Contest so far, three have gone to OCaml, three to Haskell, one to C++, and one to Cilk (Blumofe et al., 1996).

12.3 Open source: Darcs and Pugs

One of the turning points in a language’s evolution is when people start to learn it because of the applications that are written in it rather than because they are interested in the language itself. In the last few years two open-source projects, Darcs and Pugs, have started to have that effect for Haskell.

Darcs is an open-source revision-control system written in Haskell by the physicist David Roundy (Roundy, 2005). It addresses the

same challenges as the well-established incumbents such as CVS and Subversion, but its data model is very different. Rather than thinking in terms of a master repository of which users take copies, Darcs considers each user to have a fully fledged repository, with repositories exchanging updates by means of patches. This rather democratic architecture (similar to that of Arch) seems very attractive to the open-source community, and has numerous technical advantages as well (Roundy, 2005). It is impossible to say how many people use Darcs, but the user-group mailing list has 350 members, and the Darcs home page lists nearly 60 projects that use Darcs.

Darcs was originally written in C++ but, as Roundy puts it, “after working on it for a while I had an essentially solid mass of bugs” (Stosberg, 2005). He came across Haskell and, after a few experiments in 2002, rewrote Darcs in Haskell. Four years later, the source code is still a relatively compact 28,000 lines of literate Haskell (thus including the source for the 100-page manual). Roundy reports that some developers now are learning Haskell specifically in order to contribute to Darcs.

One of these programmers was Audrey Tang. She came across Darcs, spent a month learning Haskell, and jumped from there to Pierce’s book *Types and Programming Languages* (Pierce, 2002). The book suggests implementing a toy language as an exercise, so Tang picked Perl 6. At the time there were no implementations of Perl 6, at least partly because it is a ferociously difficult language to implement. Tang started her project on 1 February 2005. A year later there were 200 developers contributing to it; perhaps amazingly (considering this number) the compiler is only 18,000 lines of Haskell (including comments) (Tang, 2005). Pugs makes heavy use of parser combinators (to support a dynamically changeable parser) and several more sophisticated Haskell idioms, including GADTs (Section 6.7) and delimited continuations (Dybvig et al., 2005).

12.4 Companies using Haskell

In the commercial world, Haskell still plays only a minor role. While many Haskell programmers work for companies, they usually have an uphill battle to persuade their management to take Haskell seriously. Much of this reluctance is associated with functional programming in general, rather than Haskell in particular, although the climate is beginning to change; witness, for example, the workshops for Commercial Users of Functional Programming, held annually at ICFP since 2004. We invited four companies that use Haskell regularly to write about their experience. Their lightly edited responses constitute the rest of this section.

12.4.1 Galois Connections¹⁵

The late ’90s were the heady days of Internet companies and ridiculous valuations. At just this time Launchbury, then a professor in the functional programming research group at the Oregon Graduate Institute, began to wonder: can we do something with functional languages, and with Haskell in particular? He founded Galois Connections Inc, a company that began with the idea of finding clients for whom they could build great solutions simply by using the power of Haskell. The company tagline reflected this: Galois Connections, *Purely Functional*.

Things started well for Galois. Initial contracts came from the U.S. government for building a domain-specific language for cryptography, soon to be followed by contracts with local industry. One of these involved building a code translator for test program for chip testing equipment. Because this was a C-based problem, the Galois engineers shifted to ML, to leverage the power of the ML C-Kit

¹⁵This section is based on material contributed by John Launchbury of Galois Connections.

library. In a few months, a comprehensive code translation tool was built and kept so precisely to a compressed code-delivery schedule that the client was amazed.

From a language perspective, there were no surprises here: compilers and other code translation are natural applications for functional languages, and the abstraction and non-interference properties of functional languages meant that productivity was very high, even with minimal project management overhead. There were business challenges, however: a “can do anything” business doesn’t get known for doing anything. It has to resell its capabilities from the ground up on every sale. Market focus is needed.

Galois selected a focus area of high-confidence software, with special emphasis on information assurance. This was seen as a growth area and one in which the U.S. government already had major concerns, both for its own networks and for the public Internet. It also appeared to present significant opportunity for introducing highly innovative approaches. In this environment Haskell provided something more than simple productivity. Because of referential transparency, Haskell programs can be viewed as executable mathematics, as equations over the category of complete partial orders. In principle, at least, the specification *becomes* the program.

Examples of Haskell projects at Galois include: development tools for Cryptol, a domain-specific language for specifying cryptographic algorithms; a debugging environment for a government-grade programmable crypto-coprocessor; tools for generating FPGA layouts from Cryptol; a high-assurance compiler for the ASN.1 data-description language; a non-blocking cross-domain file system suitable for fielding in systems with multiple independent levels of security (MILS); a WebDAV server with audit trails and logging; and a wiki for providing collaboration across distinct security levels.

12.4.2 Bluespec¹⁶

Founded in June, 2003 by Arvind (MIT), Bluespec, Inc. manufactures an industry standards-based electronic design automation (EDA) toolset that is intended to raise the level of abstraction for hardware design while retaining the ability to automatically synthesise high-quality register-transfer code without compromising speed, power or area.

The name Bluespec comes from a hardware description language by the same name, which is a key enabling technology for the company. Bluespec’s design was heavily influenced by Haskell. It is basically Haskell with some extra syntactic constructs for the term rewriting system (TRS) that describes what the hardware does. The type system has been extended with types of numeric kind. Using the class system, arithmetic can be performed on these numeric types. Their purpose is to give accurate types to things like bit vectors (instead of using lists where the sizes cannot be checked by the type checker). For example:

```
bundle :: Bit[n] -> Bit[m] -> Bit[n+m]
```

Here, n and m are type variables, but they have kind Nat, and (limited) arithmetic is allowed (and statically checked) at the type level. Bluespec is really a two-level language. The full power of Haskell is available at compile time, but almost all Haskell language constructs are eliminated by a partial evaluator to get down to the basic TRS that the hardware can execute.

¹⁶This section was contributed by Rishiyur Nikhil of Bluespec.

12.4.3 Action¹⁷

Action Technologies LLC is a company with some nine employees, based in Columbus, Ohio, USA. The company specialises in artificial intelligence software for decision support.

In 2001 Action was about to begin a significant new software development project. They chose Haskell, because of its rich static type system, open-source compilers, and its active research community. At the time, no one at Action was an experienced Haskell programmer, though some employees had some experience with ML and Lisp.

Overall, their experience was extremely positive, and they now use Haskell for all their software development except for GUIs (where they use Java). They found that Haskell allows them to write succinct but readable code for rapid prototypes. As Haskell is a very high-level language, they find they can concentrate on the problem at hand without being distracted by all the attendant programming boilerplate and housekeeping. Action does a lot of research and invention, so efficiency in prototyping is very important. Use of Haskell has also helped the company to hire good programmers: it takes some intelligence to learn and use Haskell, and Action’s rare use of such an agreeable programming language promotes employee retention.

The main difficulty that Action encountered concerns efficiency: how to construct software that uses both strict and lazy evaluation well. Also, there is an initial period of difficulty while one learns what sorts of bugs evoke which incomprehensible error messages. And, although Action has been able to hire largely when they needed to, the pool of candidates with good Haskell programming skills is certainly small. A problem that Action has not yet encountered, but fears, is that a customer may object to the use of Haskell because of its unfamiliarity. (Customers sometimes ask the company to place source code in escrow, so that they are able to maintain the product if Action is no longer willing or able to do so.)

12.4.4 Linspire¹⁸

Linspire makes a Linux distribution targeted for the consumer market. The core OS team settled in 2006 on Haskell as the preferred choice for systems programming. This is an unusual choice. In this domain, it is much more common to use a combination of several shells and script languages (such as bash, awk, sed, Perl, Python). However, the results are often fragile and fraught with *ad hoc* conventions. Problems that are not solved directly by the shell are handed off to a bewildering array of tools, each with its own syntax, capabilities and shortcomings.

While not as specialised, Haskell has comparable versatility but promotes much greater uniformity. Haskell’s interpreters provide sufficient interactivity for constructing programs quickly; its libraries are expanding to cover the necessary diversity with truly reusable algorithms; and it has the added benefit that transition to compiled programs is trivial. The idioms for expressing systems programming are not quite as compact as in languages such as Perl, but this is an active area of research and the other language benefits outweigh this lack.

Static type-checking has proved invaluable, catching many errors that might have otherwise occurred in the field, especially when the cycle of development and testing is spread thin in space and time. For example, detecting and configuring hardware is impossible to test fully in the lab. Even if it were possible to collect all the

¹⁷This section was contributed by Mark Carroll of Action.

¹⁸This section was contributed by Clifford Besher of Linspire.

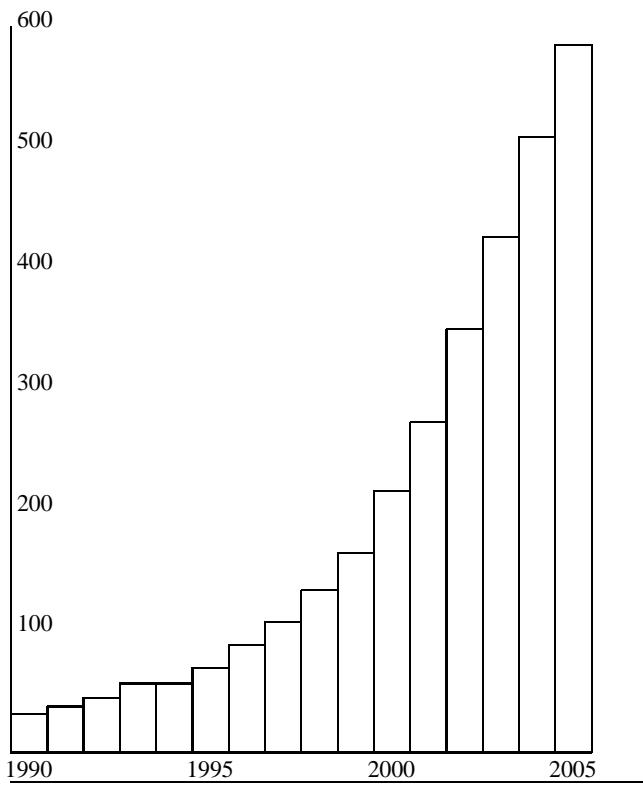


Figure 7. Growth of the “hard-core” Haskell community

various components, the time to assemble and test all the possible combinations is prohibitive. Another example is that Linspire’s tools must handle legacy data formats. Explicitly segregating these formats into separate data types prevented the mysterious errors that always seem to propagate through shell programs when the format changes.

Runtime efficiency can be a problem, but the Haskell community has been addressing this aggressively. In particular, the recent development of the `Data.ByteString` library fills the most important gap. Linspire recently converted a parser to use this module, reducing memory requirements by a factor of ten and increasing speed to be comparable with the standard command `cat`.

Learning Haskell is not a trivial task, but the economy of expression and the resulting readability seem to provide a calm inside the storm. The language, libraries and culture lead to solutions that feel like minimal surfaces: simple expressions that comprise significant complexity, with forms that seem natural, recurring in problem after problem. Open source software remains somewhat brittle, relying on the fact that most users are developers aware of its weak points. At Linspire, Haskell offers the promise of annealing a stronger whole.

12.5 The Haskell community

A language that is over 15 years old might be expected to be entering its twilight years. Perhaps surprisingly, though, Haskell appears to be in a particularly vibrant phase at the time of writing. Its use is growing strongly and appears for the first time to show signs of breaking out of its specialist-geeky niche.

The last five years have seen a variety of new community initiatives, led by a broad range of people including some outside the academic/research community. For example:

The Haskell Workshops. The first Haskell Workshop was held in conjunction with ICFP in 1995, as a one-day forum to discuss

the language and the research ideas it had begun to spawn. Subsequent workshops were held in 1997 and 1999, after which it became an annual institution. It now has a refereed proceedings published by ACM and a steady attendance of 60-90 participants. Since there is no Haskell Committee (Section 3.7), the Haskell workshop is the only forum at which corporate decisions can be, and occasionally are, taken.

The Haskell Communities and Activities Report (HCAR). In November 2001 Claus Reinke edited the first edition of the Haskell Communities and Activities Report¹⁹, a biannual newsletter that reports on what projects are going on in the Haskell community. The idea really caught on: the first edition listed 19 authors and consisted of 20 pages; but the November 2005 edition (edited by Andres Löh) lists 96 authors and runs to over 60 pages.

The #haskell IRC channel first appeared in the late 1990s, but really got going in early 2001 with the help of Shae Erisson (aka shapr)²⁰. It has grown extremely rapidly; at the time of writing, there are typically 200 people logged into the channel at any moment, with upward of 2,000 participants over a full year. The #haskell channel has spawned a particularly successful software client called `lambdabot` (written in Haskell, of course) whose many plugins include language translation, dictionary lookup, searching for Haskell functions, a theorem prover, Darcs patch tracking, and more besides.

The Haskell Weekly News. In 2005, John Goerzen decided to help people cope with the rising volume of mailing list activity by distributing a weekly summary of the most important points—the *Haskell Weekly News*, first published on the 2nd of August²¹. The HWN covers new releases, resources and tools, discussion, papers, a “Darcs corner,” and quotes-of-the-week—the latter typically being “in” jokes such as “Haskell separates Church and state.”

The Monad Reader. Another recent initiative to help a wider audience learn about Haskell is Shae Erisson’s *The Monad Reader*²², a web publication that first appeared in March 2005. The first issue declared: “*There are plenty of academic papers about Haskell, and plenty of informative pages on the Haskell Wiki. But there’s not much between the two extremes. The Monad.Reader aims to fit in there; more formal than a Wiki page, but less formal than a journal article.*” Five issues have already appeared, with many articles by practitioners, illustrated with useful code fragments.

Planet Haskell is a site for Haskell bloggers²³, started by Antti Juhani Kaijanaho in 2006.

The Google Summer of Code ran for the first time in 2005, and included just one Haskell project, carried out by Paolo Martini. Fired by his experience, Martini spearheaded a much larger Haskell participation in the 2006 Summer of Code. He organised a panel of 20 mentors, established `haskell.org` as a mentoring organisation, and attracted an astonishing 114 project proposals, of which nine were ultimately funded²⁴.

It seems clear from all this that the last five years has seen particularly rapid growth. To substantiate our gut feel, we carried out an

¹⁹ <http://haskell.org/communities/>

²⁰ http://haskell.org/haskellwiki/IRC_channel

²¹ <http://sequence.complete.org/hwn>

²² <http://www.haskell.org/hawiki/TheMonadReader>

²³ <http://planet.haskell.org>

²⁴ <http://hackage.haskell.org/trac/summer-of-code>

informal survey of the Haskell community via the Haskell mailing list, and obtained almost 600 responses from 40 countries. Clearly, our respondents belong to a self-selected group who are sufficiently enthusiastic about the language itself to follow discussion on the list, and so are not representative of Haskell users in general. In particular, it is clear from the responses that the majority of students currently being taught Haskell did not reply. Nevertheless, as a survey of the “hard core” of the community, the results are interesting.

We asked respondents when they first learnt Haskell, so we could estimate how the size of the community has changed over the years²⁵. The results are shown in Figure 7, where the bars show the total number of respondents who had learnt Haskell by the year in question. Clearly the community has been enjoying much stronger growth since 1999. This is the year that the Haskell 98 standard was published—the year that Haskell took the step from a frequently changing vehicle for research to a language with a guarantee of long-term stability. It is tempting to conclude that this is cause and effect.

Further indications of rapid growth come from mailing list activity. While the “official” Haskell mailing list has seen relatively flat traffic, the “Haskell Café” list, started explicitly in October 2000 as a forum for beginners’ questions and informal discussions, has seen traffic grow by a factor of six between 2002 and 2005. The Haskell Café is most active in the winters: warm weather seems to discourage discussion of functional programming²⁶!

Our survey also revealed a great deal about who the hard-core Haskell programmers are. One lesson is that Haskell is a programming language for the whole family—the oldest respondent was 80 years old, and the youngest just 16! It is sobering to realise that Haskell was conceived before its youngest users. Younger users do predominate, though: respondents’ median age was 27, some 25% were 35 or over, and 25% were 23 or younger.

Surprisingly, given the importance we usually attach to university teaching for technology transfer, only 48% of respondents learned Haskell as part of a university course. The majority of our respondents discovered the language by other means. Only 10% of respondents learnt Haskell as their first programming language (and 7% as their second), despite the efforts that have been made to promote Haskell for teaching introductory programming²⁷. Four out of five hard-core Haskell users were already experienced programmers by the time they learnt the language.

Haskell is still most firmly established in academia. Half of our respondents were students, and a further quarter employed in a university. 50% were using Haskell as part of their studies and 40% for research projects, so our goals of designing a language suitable for teaching and research have certainly been fulfilled. But 22% of respondents work in industry (evenly divided between large and small companies), and 10% of respondents are using Haskell for product development, so our goal of designing a language suitable for applications has also been fulfilled. Interestingly, 22% are using Haskell for open-source projects, which are also applications. Perhaps open-source projects are less constrained in the choice of programming language than industrial projects are.

The country with the most Haskell enthusiasts is the United States (115), closely followed by Portugal (91) and Germany (85). Traditional “hotbeds of functional programming” come lower down:

²⁵ Of course, this omits users who learnt Haskell but then stopped using it before our survey.

²⁶ This may explain its relative popularity in Scandinavia.

²⁷ Most Haskell textbooks are aimed at introductory programming courses.

the UK is in fourth place (49), and Sweden in sixth (29). Other countries with 20 or more respondents were the Netherlands (42) and Australia (25). It is curious that France has only six, whereas Germany has 85—perhaps French functional programmers prefer OCaml.

The picture changes, though, when we consider the proportion of Haskell enthusiasts in the general population. Now the Cayman Islands top the chart, with one Haskell enthusiast per 44,000 people. Portugal comes second, with one in 116,000, then Scandinavia—Iceland, Finland, and Sweden all have around one Haskeller per 300,000 inhabitants. In the UK, and many other countries, Haskell enthusiasts are truly “one in a million.” The United States falls between Bulgaria and Belgium, with one Haskeller for every 2,500,000 inhabitants.

If we look instead at the density of Haskell enthusiasts per unit of land mass, then the Cayman Islands are positively crowded: each Haskeller has only 262 square kilometres to program in. In Singapore, Haskellers have a little more room, at 346 square kilometres, while in the Netherlands and Portugal they have 1,000 square kilometres each. Other countries offer significantly more space—over a million square kilometres each in India, Russia, and Brazil.

12.6 Influence on other languages

Haskell has influenced several other programming languages. In many cases it is hard to ascertain whether there is a *causal* relationship between the features of a particular language and those of Haskell, so we content ourselves with mentioning similarities.

Clean is a lazy functional programming language, like Miranda and Haskell, and it bears a strong resemblance to both of these (Brus et al., 1987). Clean has adopted type classes from Haskell, but instead of using monads for input-output it uses an approach based on uniqueness (or linear) types (Achten et al., 1992).

Mercury is a language for logic programming with declared types and modes (Somogyi et al., 1996). It is influenced by Haskell in a number of ways, especially its adoption of type classes. *Hal*, a language for constraint programming built on top of Mercury, uses type classes in innovative ways to permit use of multiple constraint solvers (de la Banda et al., 2002).

Curry is a language for functional-logic programming (Hanus et al., 1995). As its name indicates, it is intended as a sort of successor to Haskell, bringing together researchers working on functional-logic languages in the same way that Haskell brought together researchers working on lazy languages. *Escher* is another language for functional-logic programming (Lloyd, 1999). Both languages have a syntax influenced by Haskell and use monads for input-output.

Cayenne is a functional language with fully fledged dependent types, designed and implemented by Lennart Augustsson (Augustsson, 1998). Cayenne is explicitly based on Haskell, although its type system differs in fundamental ways. It is significant as the first example of integrating the full power of dependent types into a programming language.

Isabelle is a theorem-proving system that makes extensive use of type classes to structure proofs (Paulson, 2004). When a type class is declared one associates with it the laws obeyed by the operations in a class (for example, that plus, times, and negation form a ring), and when an instance is declared one must prove that the instance satisfies those properties (for example, that the integers are a ring).

Python is a dynamically typed language for scripting (van Rossum, 1995). Layout is significant in Python, and it has also adopted the

list comprehension notation. In turn, *Javascript*, another dynamically typed language for scripting, is planned to adopt list comprehensions from Python, but called array comprehensions instead.

Java. The generic type system introduced in Java 5 is based on the Hindley-Milner type system (introduced in ML, and promoted by Miranda and Haskell). The use of bounded types in that system is closely related to type classes in Haskell. The type system is based on GJ, of which Wadler is a codesigner (Bracha et al., 1998).

C# and Visual Basic. The LINQ (Language INtegrated Query) features of C# 3.0 and Visual Basic 9.0 are based on monad comprehensions from Haskell. Their inclusion is due largely to the efforts of Erik Meijer, a member of the Haskell Committee, and they were inspired by his previous attempts to apply Haskell to build web applications (Meijer, 2000).

Scala. Scala is a statically typed programming language that attempts to integrate features of functional and object-oriented programming (Odersky et al., 2004; Odersky, 2006). It includes for comprehensions that are similar to monad comprehensions, and view bounds and implicit parameters that are similar to type classes.

We believe the most important legacy of Haskell will be how it influences the languages that succeed it.

12.7 Current developments

Haskell is currently undergoing a new revision. At the 2005 Haskell Workshop, Launchbury called for the definition of “Industrial Haskell” to succeed Haskell 98. So many extensions have appeared since the latter was defined that few real programs adhere to the standard nowadays. As a result, it is awkward for users to say exactly what language their application is written in, difficult for tool builders to know which extensions they should support, and impossible for teachers to know which extensions they should teach. A new standard, covering the extensions that are heavily used in industry, will solve these problems—for the time being at least. A new committee has been formed to design the new language, appropriately named Haskell’ (Haskell-prime), and the Haskell community is heavily engaged in public debate on the features to be included or excluded. When the new standard is complete, it will give Haskell a form that is tempered by real-world use.

Much energy has been spent recently on performance. One light-hearted sign of that is Haskell’s ranking in the Great Computer Language Shootout²⁸. The shootout is a benchmarking web site where over thirty language implementations compete on eighteen different benchmarks, with points awarded for speed, memory efficiency, and concise code. Anyone can upload new versions of the benchmark programs to improve their favourite language’s ranking, and early in 2006 the Haskell community began doing just that. To everyone’s amazement, despite a rather poor initial placement, on the 10th of February 2006 Haskell and GHC occupied the first place on the list! Although the shootout makes no pretence to be a scientific comparison, this does show that competitive performance is now achievable in Haskell—the inferiority complex over performance that Haskell users have suffered for so long seems now misplaced.

Part of the reason for this lies in the efficient new libraries that the growing community is developing. For example, *Data.ByteString* (by Coutts, Stewart and Leshchinskiy) represents strings as byte vectors rather than lists of characters, providing the same interface but running between one and two orders of magnitude faster. It achieves this partly thanks to an efficient representation, but also by using GHC’s rewrite rules to program the compiler’s optimiser,

so that loop fusion is performed when bytestring functions are composed. The correctness of the rewrite rules is crucial, so it is tested by QuickCheck properties, as is agreement between corresponding bytestring and *String* operations. This is a great example of using Haskell’s advanced features to achieve good performance and reliability without compromising elegance.

We interpret these as signs that, eighteen years after it was christened, Haskell is maturing. It is becoming more and more suitable for real-world applications, and the Haskell community, while still small in absolute terms, is growing strongly. We hope and expect to see this continue.

13. Conclusion

Functional programming, particularly in its purely functional form, is a radical and principled attack on the challenge of writing programs that work. It was precisely this quirky elegance that attracted many of us to the field. Back in the early ’80s, purely functional languages might have been radical and elegant, but they were also laughably impractical: they were slow, took lots of memory, and had no input/output. Things are very different now! We believe that Haskell has contributed to that progress, by sticking remorselessly to the discipline of purity, and by building a critical mass of interest and research effort behind a single language.

Purely functional programming is not necessarily the Right Way to write programs. Nevertheless, beyond our instinctive attraction to the discipline, many of us were consciously making a long-term bet that principled control of effects would ultimately turn out to be important, despite the dominance of effects-by-default in mainstream languages.

Whether that bet will truly pay off remains to be seen. But we can already see convergence. At one end, the purely functional community has learnt both the merit of effects, and at least one way to tame them. At the other end, mainstream languages are adopting more and more declarative constructs: comprehensions, iterators, database query expressions, first-class functions, and more besides. We expect this trend to continue, driven especially by the goad of parallelism, which punishes unrestricted effects cruelly.

One day, Haskell will be no more than a distant memory. But we believe that, when that day comes, the ideas and techniques that it nurtured will prove to have been of enduring value through their influence on languages of the future.

14. Acknowledgements

The Haskell community is open and vibrant, and many, many people have contributed to the language design beyond those mentioned in our paper.

The members of the Haskell Committee played a particularly important role, however. Here they are, with their affiliations during the lifetime of the committee, and identifying those who served as Editor for some iteration of the language: Arvind (MIT), Lennart Augustsson (Chalmers University), Dave Barton (Mitre Corp), Richard Bird (University of Oxford), Brian Boutel (Victoria University of Wellington), Warren Burton (Simon Fraser University), Jon Fairbairn (University of Cambridge), Joseph Fasel (Los Alamos National Laboratory), Andy Gordon (University of Cambridge), Maria Guzman (Yale University), Kevin Hammond [editor] (University of Glasgow), Ralf Hinze (University of Bonn), Paul Hudak [editor] (Yale University), John Hughes [editor] (University of Glasgow, Chalmers University), Thomas Johnsson (Chalmers University), Mark Jones (Yale University, University of Nottingham, Oregon Graduate Institute), Dick Kieburz (Oregon Graduate

²⁸ See <http://shootout.alioth.debian.org>

Institute), John Launchbury (University of Glasgow, Oregon Graduate Institute), Erik Meijer (Utrecht University), Rishiyur Nikhil (MIT), John Peterson [editor] (Yale University), Simon Peyton Jones [editor] (University of Glasgow, Microsoft Research Ltd), Mike Reeve (Imperial College), Alastair Reid (University of Glasgow, Yale University), Colin Runciman (University of York), Philip Wadler [editor] (University of Glasgow), David Wise (Indiana University), and Jonathan Young (Yale University).

We also thank those who commented on a draft of this paper, or contributed their recollections: Thiago Arrais, Lennart Augustsson, Dave Bayer, Alistair Bayley, Richard Bird, James Bostock, Warren Burton, Paul Callahan, Michael Cartmell, Robert Dockins, Susan Eisenbach, Jon Fairbairn, Tony Field, Jeremy Gibbons, Kevin Glynn, Kevin Hammond, Graham Hutton, Johan Jeuring, Thomas Johnsson, Mark Jones, Jevgeni Kabanov, John Kraemer, Ralf Lämmel, Jan-Willem Maessen, Michael Mahoney, Ketil Malde, Evan Martin, Paolo Martini, Conor McBride, Greg Michaelson, Neil Mitchell, Ben Moseley, Denis Moskvin, Russell O'Connor, Chris Okasaki, Rex Page, Andre Pang, Will Partain, John Peterson, Benjamin Pierce, Bernie Pope, Greg Restall, Alberto Ruiz, Colin Runciman, Kostis Sagonas, Andres Sicard, Christian Sievers, Ganesh Sittampalam, Don Stewart, Joe Stoy, Peter Stuckey, Martin Sulzmann, Josef Svenningsson, Simon Thompson, David Turner, Jared Updike, Michael Vanier, Janis Voigtlander, Johannes Waldmann, Malcolm Wallace, Mitchell Wand, Eric Willigers, and Marc van Woerkom.

Some sections of this paper are based directly on material contributed by Lennart Augustsson, Clifford Bessers, Paul Callaghan, Mark Carroll, Mark Jones, John Launchbury, Rishiyur Nikhil, David Roundy, Audrey Tang, and David Turner. We thank them very much for their input. We would also like to give our particular thanks to Bernie Pope and Don Stewart, who prepared the time line given in Figure 2.

Finally, we thank the program committee and referees of HOPL III.

References

- Achten, P. and Peyton Jones, S. (2000). Porting the Clean Object I/O library to Haskell. In Mohnen, M. and Koopman, P., editors, *Proceedings of the 12th International Workshop on the Implementation of Functional Languages, Aachen (IFL'00), selected papers*, number 2011 in Lecture Notes in Computer Science, pages 194–213. Springer.
- Achten, P. and Plasmeijer, R. (1995). The ins and outs of clean I/O. *Journal of Functional Programming*, 5(1):81–110.
- Achten, P., van Groningen, J., and Plasmeijer, M. (1992). High-level specification of I/O in functional languages. In (Launchbury and Sansom, 1992), pages 1–17.
- Angelov, K. and Marlow, S. (2005). Visual Haskell: a full-featured Haskell development environment. In *Proceedings of ACM Workshop on Haskell, Tallinn*, Tallinn, Estonia. ACM.
- Appel, A. and MacQueen, D. (1987). A standard ML compiler. In Kahn, G., editor, *Proceedings of the Conference on Functional Programming and Computer Architecture, Portland*. LNCS 274, Springer Verlag.
- Arts, T., Hughes, J., Johansson, J., and Wiger, U. (2006). Testing telecoms software with quviq quickcheck. In Trinder, P., editor, *ACM SIGPLAN Erlang Workshop*, Portland, Oregon. ACM SIGPLAN.
- Arvind and Nikhil, R. (1987). Executing a program on the MIT tagged-token dataflow architecture. In *Proc PARLE (Parallel Languages and Architectures, Europe) Conference, Eindhoven*. Springer Verlag LNCS.
- Atkins, D., Ball, T., Bruns, G., and Cox, K. (1999). Mawl: A domain-specific language for form-based services. *IEEE Transactions on Software Engineering*, 25(3):334–346.
- Augustsson, L. (1984). A compiler for lazy ML. In (LFP84, 1984), pages 218–227.
- Augustsson, L. (1998). Cayenne — a language with dependent types. In (ICFP98, 1998), pages 239–250.
- Baars, A., Lh, A., and Swierstra, D. (2004). Parsing permutation phrases. *Journal of Functional Programming*, 14:635–646.
- Baars, A. L. and Swierstra, S. D. (2002). Typing dynamic typing. In (ICFP02, 2002), pages 157–166.
- Backus, J. (1978a). Can programming be liberated from the von Neumann style? *Communications of the ACM*, 21(8).
- Backus, J. (1978b). Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–41.
- Barendsen, E. and Smetsers, S. (1996). Uniqueness typing for functional languages with graph rewriting semantics. *Mathematical Structures in Computer Science*, 6:579–612.
- Barron, D., Buxton, J., Hartley, D., Nixon, E., and Strachey, C. (1963). The main features of cpl. *The Computer Journal*, 6(2):134–143.
- Barth, P., Nikhil, R., and Arvind (1991). M-structures: extending a parallel, non-strict functional language with state. In Hughes, R., editor, *ACM Conference on Functional Programming and Computer Architecture (FPCA'91)*, volume 523 of *Lecture Notes in Computer Science*, pages 538–568. Springer Verlag, Boston.
- Barton, D. (1995). Advanced modeling features of MHDL. In *Proceedings of International Conference on Electronic Hardware Description Languages*.
- Bird, R. and Paterson, R. (1999). De Bruijn notation as a nested datatype. *Journal of Functional Programming*, 9(1):77–91.
- Bird, R. and Wadler, P. (1988). *Introduction to Functional Programming*. Prentice Hall.
- Bjesse, P., Claessen, K., Sheeran, M., and Singh, S. (1998). Lava: Hardware design in haskell. In *International Conference on Functional Programming*, pages 174–184.
- Bloss, A. (1988). *Path Analysis: Using Order-of-Evaluation Information to Optimize Lazy Functional Languages*. PhD thesis, Yale University, Department of Computer Science.
- Bloss, A., Hudak, P., and Young, J. (1988a). Code optimizations for lazy evaluation. *Lisp and Symbolic Computation: An International Journal*, 1(2):147–164.
- Bloss, A., Hudak, P., and Young, J. (1988b). An optimizing compiler for a modern functional language. *The Computer Journal*, 31(6):152–161.
- Blott, S. (1991). *Type Classes*. PhD thesis, Department of Computing Science, Glasgow University.
- Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H., and Zhou, Y. (1996). Cilk: An efficient multi-threaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69.
- Boquist, U. (1999). *Code Optimisation Techniques for Lazy Functional Languages*. PhD thesis, Chalmers University of Technology.

- nology, Sweden.
- Bracha, G., Odersky, M., Stoutamire, D., and Wadler, P. (1998). Making the future safe for the past: Adding genericity to the Java programming language. In Chambers, C., editor, *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 183–200, Vancouver, BC.
- Brus, T., van Eckelen, M., van Leer, M., and Plasmeijer, M. (1987). Clean — a language for functional graph rewriting. In Kahn, G., editor, *Functional Programming Languages and Computer Architecture*, pages 364–384. LNCS 274, Springer Verlag.
- Burge, W. (1975). *Recursive Programming Techniques*. Addison Wesley.
- Burstall, R. (1969). Proving properties of programs by structural induction. *The Computer Journal*, pages 41–48.
- Burstall, R. (1977). Design considerations for a functional programming language. In *The Software Revolution*. Infotech.
- Burstall, R. and Darlington, J. (1977). A transformation system for developing recursive programs. *JACM*, 24(1):44–67.
- Burstall, R. M., MacQueen, D. B., and Sannella, D. T. (1980). HOPE: An experimental applicative language. In *Conference Record of the 1980 LISP Conference*, pages 136–143.
- Burton, W., Meijer, E., Sansom, P., Thompson, S., and Wadler, P. (1996). Views: An extension to Haskell pattern matching, <http://haskell.org/development/views.html>.
- Callaghan, P. (1998). *An Evaluation of LOLITA and Related Natural Language Processing Systems*. PhD thesis, Department of Computer Science, University of Durham.
- Carlier, S. and Bobbio, J. (2004). hop.
- Carlson, W., Hudak, P., and Jones, M. (1993). An experiment using Haskell to prototype “geometric region servers” for Navy command and control. Research Report 1031, Department of Computer Science, Yale University.
- Carlsson, M. and Hallgren, T. (1993). Fudgets — a graphical user interface in a lazy functional language. In (FPCA93, 1993), pages 321–330.
- Chakravarty, M. (1999a). C → Haskell: yet another interfacing tool. In Koopman, P. and Clack, C., editors, *International Workshop on Implementing Functional Languages (IFL'99)*, number 1868 in Lecture Notes in Computer Science, Lochem, The Netherlands. Springer Verlag.
- Chakravarty, M. (1999b). Lazy lexing is fast. In Middeldorp, A. and Sato, T., editors, *Fourth Fuji International Symposium on Functional and Logic Programming*, Lecture Notes in Computer Science. Springer Verlag.
- Chakravarty, M., editor (2002). *Proceedings of the 2002 Haskell Workshop, Pittsburgh*.
- Chakravarty, M., Keller, G., and Peyton Jones, S. (2005a). Associated type synonyms. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'05)*, Tallinn, Estonia.
- Chakravarty, M., Keller, G., Peyton Jones, S., and Marlow, S. (2005b). Associated types with class. In *ACM Symposium on Principles of Programming Languages (POPL'05)*. ACM Press.
- Chen, K., Hudak, P., and Odersky, M. (1992). Parametric type classes. In *Proceedings of ACM Conference on Lisp and Functional Programming*, pages 170–181. ACM.
- Cheney, J. and Hinze, R. (2003). First-class phantom types. CUCIS TR2003-1901, Cornell University.
- Cheong, M. H. (2005). *Functional Programming and 3D Games*. Undergraduate thesis, University of New South Wales.
- Church, A. (1941). The calculi of lambda-conversion. *Annals of Mathematics Studies*, 6.
- Claessen, K. (2004). Parallel parsing processes. *Journal of Functional Programming*, 14:741–757.
- Claessen, K. and Hughes, J. (2000). QuickCheck: a lightweight tool for random testing of Haskell programs. In (ICFP00, 2000), pages 268–279.
- Claessen, K. and Hughes, J. (2002). Testing monadic code with QuickCheck. In (Chakravarty, 2002).
- Claessen, K. and Sands, D. (1999). Observable sharing for functional circuit description. In Thiagarajan, P. and Yap, R., editors, *Advances in Computing Science (ASIAN'99); 5th Asian Computing Science Conference*, Lecture Notes in Computer Science, pages 62–73. Springer Verlag.
- Cooper, G. and Krishnamurthi, S. (2006). Embedding dynamic dataflow in a call-by-value language. In *15th European Symposium on Programming*, volume 3924 of *LNCS*. Springer Verlag.
- Courtney, A. (2004). *Modelling User Interfaces in a Functional Language*. PhD thesis, Department of Computer Science, Yale University.
- Courtney, A. and Elliott, C. (2001). Genuinely functional user interfaces. In *Proc. of the 2001 Haskell Workshop*, pages 41–69.
- Curry, H. and Feys, R. (1958). *Combinatory Logic, Vol. I*. North-Holland, Amsterdam.
- Damas, L. and Milner, R. (1982). Principal type-schemes for functional programs. In *Conference Record of the 9th Annual ACM Symposium on Principles of Programming Languages*, pages 207–12, New York. ACM Press.
- Danielsson, N. A., Hughes, J., Jansson, P., and Gibbons, J. (2006). Fast and loose reasoning is morally correct. *SIGPLAN Not.*, 41(1):206–217.
- Darlington, J., Henderson, P., and Turner, D. (1982). *Advanced Course on Functional Programming and its Applications*. Cambridge University Press.
- Darlington, J. and Reeve, M. (1981). ALICE — a multiprocessor reduction machine for the parallel evaluation of applicative languages. In *Proc Conference on Functional Programming Languages and Computer Architecture, Portsmouth, New Hampshire*, pages 66–76. ACM.
- Davis, A. (1977). The architecture of ddm1: a recursively structured data driven machine. Technical Report UUCS-77-113, University of Utah.
- de la Banda, M. G., Demoen, B., Marriott, K., and Stuckey, P. (2002). To the gates of HAL: a HAL tutorial. In *Proceedings of the Sixth International Symposium on Functional and Logic Programming*. Springer Verlag LNCS 2441.
- Diatchki, I., Jones, M., and Hallgren, T. (2002). A formal specification of the Haskell 98 module system. In (Chakravarty, 2002).
- Dijkstra, E. (1981). Trip report E.W. Dijkstra, Newcastle, 19-25 July 1981. Dijkstra working note EWD798.

- Dybjer, P. (1991). Inductive sets and families in Martin-Löf's type theory. In Huet, G. and Plotkin, G., editors, *Logical Frameworks*. Cambridge University Press.
- Dybvig, K., Peyton Jones, S., and Sabry, A. (2005). A monadic framework for delimited continuations. To appear in the *Journal of Functional Programming*.
- Elliott, C. (1996). A brief introduction to activevrml. Technical Report MSR-TR-96-05, Microsoft Research.
- Elliott, C. (1997). Modeling interactive 3D and multimedia animation with an embedded language. In *Proceedings of the first conference on Domain-Specific Languages*, pages 285–296. USENIX.
- Elliott, C. and Hudak, P. (1997). Functional reactive animation. In *International Conference on Functional Programming*, pages 263–273.
- Elliott, C., Schechter, G., Yeung, R., and Abi-Ezzi, S. (1994). Tbag: A high level framework for interactive, animated 3d graphics applications. In *Proceedings of SIGGRAPH '94*, pages 421–434. ACM SIGGRAPH.
- Ennals, R. and Peyton Jones, S. (2003). Optimistic evaluation: an adaptive evaluation strategy for non-strict programs. In (ICFP03, 2003).
- Evans, A. (1968). Pal—a language designed for teaching programming linguistics. In *Proceedings ACM National Conference*.
- Fairbairn, J. (1982). Ponder and its type system. Technical Report TR-31, Cambridge University Computer Lab.
- Fairbairn, J. (1985). Design and implementation of a simple typed language based on the lambda-calculus. Technical Report 75, University of Cambridge Computer Laboratory.
- Faxen, K.-F. (2002). A static semantics for Haskell. *Journal of Functional Programming*, 12(4&5).
- Field, A., Hunt, L., and While, R. (1992). The semantics and implementation of various best-fit pattern matching schemes for functional languages. Technical Report Doc 92/13, Dept of Computing, Imperial College.
- Finne, S., Leijen, D., Meijer, E., and Peyton Jones, S. (1998). H/Direct: a binary foreign language interface for Haskell. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, volume 34(1) of *ACM SIGPLAN Notices*, pages 153–162. ACM Press, Baltimore.
- Finne, S. and Peyton Jones, S. (1995). Composing Haggis. In *Proc 5th Eurographics Workshop on Programming Paradigms in Graphics, Maastricht*.
- Ford, B. (2002). Packrat parsing: simple, powerful, lazy, linear time. In (ICFP02, 2002), pages 36–47.
- FPCA93 (1993). *ACM Conference on Functional Programming and Computer Architecture (FPCA'93)*, Copenhagen. ACM.
- FPCA95 (1995). *ACM Conference on Functional Programming and Computer Architecture (FPCA'95)*, La Jolla, California. ACM.
- Friedman, D. and Wise, D. (1976). CONS should not evaluate its arguments. *Automata, Languages, and Programming*, pages 257–281.
- Frost, R. (2006). Realization of natural-language interfaces using lazy functional programming. *ACM Computing Surveys*, 38(4). Article No. 11.
- Gaster, B. (1998). *Records, Variants, and Qualified Types*. PhD thesis, Department of Computer Science, University of Nottingham.
- Gaster, B. R. and Jones, M. P. (1996). A polymorphic type system for extensible records and variants. Technical Report TR-96-3, Department of Computer Science, University of Nottingham.
- Gill, A. (2000). Debugging Haskell by observing intermediate data structures. In *Haskell Workshop*. ACM SIGPLAN.
- Gill, A., Launchbury, J., and Peyton Jones, S. (1993). A short cut to deforestation. In *ACM Conference on Functional Programming and Computer Architecture (FPCA'93)*, pages 223–232, Copenhagen. ACM Press. ISBN 0-89791-595-X.
- Girard, J.-Y. (1990). The system F of variable types: fifteen years later. In Huet, G., editor, *Logical Foundations of Functional Programming*. Addison-Wesley.
- Glynn, K., Stuckey, P., and Sulzmann, M. (2000). Type classes and constraint handling rules. In *First Workshop on Rule-Based Constraint Reasoning and Programming*.
- Gödel, K. (1931). Über formal unentscheidbare sätze der principia mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38:173–198. Pages 596–616 of (van Heijenoort, 1967).
- Gordon, M., Milner, R., and Wadsworth, C. (1979). *Edinburgh LCF*. Springer Verlag LNCS 78.
- Graham, P. (2004). Beating the averages. In *Hackers and Painters*. O'Reilly.
- Graunke, P., Krishnamurthi, S., Hoeven, S. V. D., and Felleisen, M. (2001). Programming the web with high-level programming languages. In *Proceedings 10th European Symposium on Programming*, pages 122–136. Springer Verlag LNCS 2028.
- Hall, C. and O'Donnell, J. (1985). Debugging in a side-effect-free programming environment. In *Proc ACM Symposium on Language Issues and Programming Environments*. ACM, Seattle.
- Hallgren, T. (2001). Fun with functional dependencies. In *Proc Joint CS/CE Winter Meeting, Chalmers University, Varberg, Sweden*.
- Hallgren, T., Jones, M. P., Leslie, R., and Tolmach, A. (2005). A principled approach to operating system construction in Haskell. In *ICFP '05: Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, pages 116–128, New York, NY, USA. ACM Press.
- Hanus, M., Kuchen, H., and Moreno-Navarro, J. (1995). Curry: A truly functional logic language. In *Proceedings of the ILPS '95 Postconference Workshop on Visions for the Future of Logic Programming*.
- Harris, T., Marlow, S., Peyton Jones, S., and Herlihy, M. (2005). Composable memory transactions. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP'05)*.
- Harrison, W. and Kamin, S. (1998). Modular compilers based on monad transformers. In *Proc International Conference on Computer Languages*, pages 122–131.
- Hartel, P., Feeley, M., Alt, M., Augustsson, L., Bauman, P., Weis, P., and Wentworth, P. (1996). Pseudoknot: a float-intensive benchmark for functional compilers. *Journal of Functional Programming*, 6(4).
- Haskell01 (2001). *Proceedings of the 2001 Haskell Workshop, Florence*.

- Haskell04 (2004). *Proceedings of ACM Workshop on Haskell*, Snowbird, Utah. ACM.
- Heeren, B., Hage, J., and Swierstra, S. (2003a). Scripting the type inference process. In (ICFP03, 2003), pages 3–14.
- Heeren, B., Leijen, D., and van IJzendoorn, A. (2003b). Helium, for learning Haskell. In *ACM Sigplan 2003 Haskell Workshop*, pages 62 – 71, New York. ACM Press.
- Henderson, P. (1982). Functional geometry. In *Proc ACM Symposium on Lisp and Functional Programming*, pages 179–187. ACM.
- Henderson, P. and Morris, J. (1976). A lazy evaluator. In *In Proceedings of 3rd International Conference on Principles of Programming Languages (POPL'76)*, pages 95–103.
- Herington, D. (2002). Hunit home page. <http://hunit.sourceforge.net>.
- Hinze, R. (2000). A new approach to generic functional programming. In (POPL00, 2000), pages 119–132.
- Hinze, R. (2001). Manufacturing datatypes. *Journal of Functional Programming*, 1.
- Hinze, R. (2003). Fun with phantom types. In Gibbons, J. and de Moor, O., editors, *The Fun of Programming*, pages 245–262. Palgrave.
- Hinze, R. (2004). Generics for the masses. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'04)*, Snowbird, Utah. ACM.
- Hinze, R., Jeuring, J., and Lh, A. (2006). Comparing approaches to generic programming in Haskell. In *Generic Programming, Advanced Lectures*, LNCS. Springer-Verlag.
- Hinze, R. and Peyton Jones, S. (2000). Derivable type classes. In Hutton, G., editor, *Proceedings of the 2000 Haskell Workshop, Montreal*. Nottingham University Department of Computer Science Technical Report NOTTCS-TR-00-1.
- Hudak, P. (1984a). ALFL Reference Manual and Programmer's Guide. Research Report YALEU/DCS/RR-322, Second Edition, Yale University, Dept. of Computer Science.
- Hudak, P. (1984b). Distributed applicative processing systems – project goals, motivation and status report. Research Report YALEU/DCS/RR-317, Yale University, Dept. of Computer Science.
- Hudak, P. (1989). Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, 21(3):359–411.
- Hudak, P. (1996a). Building domain-specific embedded languages. *ACM Computing Surveys*, 28A.
- Hudak, P. (1996b). Haskore music tutorial. In *Second International School on Advanced Functional Programming*, pages 38–68. Springer Verlag, LNCS 1129.
- Hudak, P. (1998). Modular domain specific languages and tools. In *Proceedings of Fifth International Conference on Software Reuse*, pages 134–142. IEEE Computer Society.
- Hudak, P. (2000). *The Haskell School of Expression – Learning Functional Programming Through Multimedia*. Cambridge University Press, New York.
- Hudak, P. (2003). Describing and interpreting music in Haskell. In Gibbons, J. and de Moor, O., editors, *The Fun of Programming*, chapter 4. Palgrave.
- Hudak, P. (2004). Polymorphic temporal media. In *Proceedings of PADL'04: 6th International Workshop on Practical Aspects of Declarative Languages*. Springer Verlag LNCS.
- Hudak, P., Courtney, A., Nilsson, H., and Peterson, J. (2003). Arrows, robots, and functional reactive programming. In Jeuring, J. and Jones, S. P., editors, *Advanced Functional Programming, 4th International School*, volume 2638 of *Lecture Notes in Computer Science*. Springer-Verlag.
- Hudak, P., Makucevich, T., Gadde, S., and Whong, B. (1996). Haskore music notation – an algebra of music. *Journal of Functional Programming*, 6(3):465–483.
- Hudak, P. and Sundaresh, R. (1989). On the expressiveness of purely-functional I/O systems. Research Report YALEU/DCS/RR-665, Department of Computer Science, Yale University.
- Hudak, P. and Young, J. (1986). Higher-order strictness analysis in untyped lambda calculus. In *ACM Symposium on Principles of Programming Languages*, pages 97–109.
- Huet, G. (1975). A unification algorithm for typed lambda-calculus. *Theoretical Computer Science*, 1:22–58.
- Huet, G. and Levy, J. (1979). Call by need computations in non-ambiguous linear term-rewriting systems. Report 359, INRIA.
- Hughes, J. (1989). Why functional programming matters. *The Computer Journal*, 32(2):98–107.
- Hughes, J. (1995). The design of a pretty-printing library. In Jeuring, J. and Meijer, E., editors, *Advanced Functional Programming*, pages 53–96. Springer Verlag, LNCS 925.
- Hughes, J. (2000). Generalising monads to arrows. *Science of Computer Programming*, 37:67–111.
- Hughes, R. (1983). *The Design and Implementation of Programming Languages*. Ph.D. thesis, Programming Research Group, Oxford University.
- Hutton, G. and Meijer, E. (1998). Monadic parsing in Haskell. *Journal of Functional Programming*, 8:437–444.
- ICFP00 (2000). *ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, Montreal. ACM.
- ICFP02 (2002). *ACM SIGPLAN International Conference on Functional Programming (ICFP'02)*, Pittsburgh. ACM.
- ICFP03 (2003). *ACM SIGPLAN International Conference on Functional Programming (ICFP'03)*, Uppsala, Sweden. ACM.
- ICFP97 (1997). *ACM SIGPLAN International Conference on Functional Programming (ICFP'97)*, Amsterdam. ACM.
- ICFP98 (1998). *ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, volume 34(1) of *ACM SIGPLAN Notices*, Baltimore. ACM.
- ICFP99 (1999). *ACM SIGPLAN International Conference on Functional Programming (ICFP'99)*, Paris. ACM.
- Jansson, P. and Jeuring, J. (1997). PolyP — a polytypic programming language extension. In *24th ACM Symposium on Principles of Programming Languages (POPL'97)*, pages 470–482, Paris. ACM.
- Jansson, P. and Jeuring, J. (1999). Polytypic compact printing and parsing. In *European Symposium on Programming*, volume 1576 of *Lecture Notes in Computer Science*, pages 273–287. Springer-Verlag.

- Johann, P. and Voigtlander, J. (2004). Free theorems in the presence of seq. In *ACM Symposium on Principles of Programming Languages (POPL'04)*, pages 99–110, Charleston. ACM.
- Johnson, S. (1984). *Synthesis of Digital Designs from Recursive Equations*. ACM Distinguished Dissertation. MIT Press.
- Johnsson, T. (1984). Efficient compilation of lazy evaluation. In *Proc SIGPLAN Symposium on Compiler Construction, Montreal*. ACM.
- Jones, M. (1991). Type inference for qualified types. PRG-TR-10-91, Programming Research Group, Oxford, Oxford University.
- Jones, M. (1992). A theory of qualified types. In *European Symposium on Programming (ESOP'92)*, number 582 in Lecture Notes in Computer Science, Rennes, France. Springer Verlag.
- Jones, M. (1993). A system of constructor classes: overloading and implicit higher-order polymorphism. In (FPCA93, 1993).
- Jones, M. (1994). *Qualified Types: Theory and Practice*. Cambridge University Press.
- Jones, M. (1995). Simplifying and improving qualified types. In (FPCA95, 1995).
- Jones, M. (1999). Typing Haskell in Haskell. In (Meijer, 1999). Available at <ftp://ftp.cs.uu.nl/pub/RUU/CS/techreps/CS-1999/1999-28.pdf>.
- Jones, M. (2000). Type classes with functional dependencies. In *European Symposium on Programming (ESOP'00)*, number 1782 in Lecture Notes in Computer Science, Berlin, Germany. Springer Verlag.
- Jones, M. and Duponcheel, L. (1994). Composing monads. Technical Report YALEU/DCS/RR-1004, Yale University.
- Jouannaud, J.-P., editor (1985). *ACM Conference on Functional Programming and Computer Architecture (FPCA'85)*, volume 201 of *Lecture Notes in Computer Science*, Nancy, France. Springer-Verlag.
- Kaes, S. (1988). Parametric overloading in polymorphic programming languages. In *Proceedings of the 2nd European Symposium on Programming*.
- Keller, R., Lindstrom, G., and Patil, S. (1979). A loosely coupled applicative multiprocessing system. In *AFIPS Conference Proceedings*, pages 613–622.
- Kelsey, R., Clinger, W., and Rees, J. (1998). Revised⁵ report on the algorithmic language Scheme. *SIGPLAN Notices*, 33(9):26–76.
- Kiselyov, O., Lmmel, R., and Schupke, K. (2004). Strongly typed heterogeneous collections. In (Haskell04, 2004), pages 96–107.
- Kiselyov, O. and Shan, K. (2004). Implicit configurations; or, type classes reflect the values of types. In (Haskell04, 2004), pages 33–44.
- Knuth, D. (1984). Literate programming. *Computer Journal*, 27(2):97–111.
- Kranz, D., Kelsey, R., Rees, J., Hudak, P., Philbin, J., and Adams, N. (1986). Orbit: an optimizing compiler for Scheme. In *SIGPLAN '86 Symposium on Compiler Construction*, pages 219–233. ACM. Published as SIGPLAN Notices Vol. 21, No. 7, July 1986.
- Kranz, D., Kesley, R., Rees, J., Hudak, P., Philbin, J., and Adams, N. (2004). Retrospective on: Orbit: an optimizing compiler for Scheme. *ACM SIGPLAN Notices, 20 Years of the ACM SIGPLAN Conference on Programming Language Design and Implementation (1979–1999): A Selection*, 39(4).
- Lämmel, R. and Peyton Jones, S. (2003). Scrap your boilerplate: a practical approach to generic programming. In *ACM SIGPLAN International Workshop on Types in Language Design and Implementation (TLDI'03)*, pages 26–37, New Orleans. ACM Press.
- Lämmel, R. and Peyton Jones, S. (2005). Scrap your boilerplate with class: Extensible generic functions. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'05)*, Tallinn, Estonia.
- Landin, P. (1966). The next 700 programming languages. *Communications of the ACM*, 9(3):157–166.
- Landin, P. J. (1964). The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320.
- Läufer, K. (1996). Type classes with existential types. *Journal of Functional Programming*, 6(3):485–517.
- Läufer, K. and Odersky, M. (1994). Polymorphic type inference and abstract data types. *ACM Transactions on Programming Languages and Systems*, 16(5):1411–1430.
- Launchbury, J. (1993). Lazy imperative programming. In *Proc ACM Sigplan Workshop on State in Programming Languages, Copenhagen* (available as YALEU/DCS/RR-968, Yale University), pages pp46–56.
- Launchbury, J. and Peyton Jones, S. (1995). State in Haskell. *Lisp and Symbolic Computation*, 8(4):293–342.
- Launchbury, J. and Sabry, A. (1997). Monadic state: Axiomatization and type safety. In (ICFP97, 1997), pages 227–238.
- Launchbury, J. and Sansom, P., editors (1992). *Functional Programming, Glasgow 1992*, Workshops in Computing. Springer Verlag.
- Leijen, D. and Meijer, E. (1999). Domain-specific embedded compilers. In *Proc 2nd Conference on Domain-Specific Languages (DSL'99)*, pages 109–122.
- Lewis, J., Shields, M., Meijer, E., and Launchbury, J. (2000). Implicit parameters: dynamic scoping with static types. In (POPL00, 2000).
- LFP84 (1984). *ACM Symposium on Lisp and Functional Programming (LFP'84)*. ACM.
- Li, H., Reinke, C., and Thompson, S. (2003). Tool support for refactoring functional programs. In Jeuring, J., editor, *Proceedings of the 2003 Haskell Workshop, Uppsala*.
- Liang, S., Hudak, P., and Jones, M. (1995). Monad transformers and modular interpreters. In *22nd ACM Symposium on Principles of Programming Languages (POPL'95)*, pages 333–343. ACM.
- Lindig, C. (2005). Random testing of C calling conventions. In *AADEBUG*, pages 3–12.
- Lloyd, J. W. (1999). Programming in an integrated functional and logic language. *Journal of Functional and Logic Programming*.
- Löh, A., Clarke, D., and Jeuring, J. (2003). Dependency-style Generic Haskell. In (ICFP03, 2003), pages 141–152.
- Long, D. and Garigliano, R. (1993). *Reasoning by Analogy and Causality (A Model and Application)*. Ellis Horwood.

- Lüth, C. and Ghani, N. (2002). Composing monads using coproducts. In (ICFP02, 2002), pages 133–144.
- Maessen, J.-W. (2002). Eager Haskell: Resource-bounded execution yields efficient iteration. In *The Haskell Workshop, Pittsburgh*.
- Major, F. and Turcotte, M. (1991). The combination of symbolic and numerical computation for three-dimensional modelling of RNA. *SCIENCE*, 253:1255–1260.
- Marlow, S., Peyton Jones, S., and Thaller, W. (2004). Extending the Haskell Foreign Function Interface with concurrency. In *Proceedings of Haskell Workshop, Snowbird, Utah*, pages 57–68.
- Matthews, J., Cook, B., and Launchbury, J. (1998). Microprocessor specification in Hawk. In *International Conference on Computer Languages*, pages 90–101.
- McBride, C. (2002). Faking it: Simulating dependent types in Haskell. *Journal of Functional Programming*, 12(4&5):375–392.
- McCarthy, J. L. (1960). Recursive functions of symbolic expressions and their computation by machine, Part I. *Communications of the ACM*, 3(4):184–195. The original Lisp paper.
- Meijer, E., editor (1999). *Proceedings of the 1999 Haskell Workshop*, number UU-CS-1999-28 in Technical Reports. Available at <ftp://ftp.cs.uu.nl/pub/RUU/CS/techreps/CS-1999/1999-28.pdf>.
- Meijer, E. (2000). Server side web scripting in Haskell. *Journal of Functional Programming*, 10(1):1–18.
- Meijer, E. and Claessen, K. (1997). The design and implementation of Mondrian. In Launchbury, J., editor, *Haskell Workshop*, Amsterdam, Netherlands.
- Milner, R. (1978). A theory of type polymorphism in programming. *JCSS*, 13(3).
- Milner, R. (1984). A proposal for Standard ML. In *ACM Symposium on LISP and Functional Programming*, pages 184–197.
- Milner, R. and Tofte, M. (1990). *The Definition of Standard ML*. MIT Press.
- Milner, R., Tofte, M., Harper, R., and MacQueen, D. (1997). *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, Massachusetts.
- Mitchell, J. and Plotkin, G. (1985). Abstract types have existential type. In *Twelfth Annual ACM Symposium on Principles of Programming Languages (POPL'85)*, pages 37–51.
- Moggi, E. (1989). Computational lambda calculus and monads. In *Logic in Computer Science, California*. IEEE.
- Moggi, E. (1991). Notions of computation and monads. *Information and Computation*, 93:55–92.
- Neubauer, M., Thiemann, P., Gasbichler, M., and Sperber, M. (2001). A functional notation for functional dependencies. In (Haskell01, 2001).
- Neubauer, M., Thiemann, P., Gasbichler, M., and Sperber, M. (2002). Functional logic overloading. In *ACM Symposium on Principles of Programming Languages (POPL'02)*, pages 233–244, Portland. ACM.
- Nikhil, R. S. and Arvind (2001). *Implicit Parallel Programming in pH*. Morgan Kaufman.
- Nilsson, H. and Fritzson, P. (1994). Algorithmic debugging for lazy functional languages. *Journal of Functional Programming*, 4(3):337–370.
- Nilsson, H. and Sparud, J. (1997). The evaluation dependence tree as a basis for lazy functional debugging. *Automated Software Engineering*, 4(2):121–150.
- Nordin, T., Peyton Jones, S., and Reid, A. (1997). Green Card: a foreign-language interface for Haskell. In Launchbury, J., editor, *Haskell Workshop*, Amsterdam.
- Odersky, M. (2006). Changes between Scala version 1.0 and 2.0. Technical report, EPFL Lausanne.
- Odersky, M., Altherr, P., Cremet, V., Emir, B., Maneth, S., Michelioud, S., Mihaylov, N., Schinz, M., Stenman, E., and Zenger, M. (2004). An overview of the Scala programming language. Technical Report IC/2004/640, EPFL Lausanne.
- O'Donnell, J. (1995). From transistors to computer architecture: teaching functional circuit specification in Hydra. In *Symposium on Functional Programming Languages in Education*, volume 1022 of *LNCS*. Springer-Verlag.
- Ohori, A. (1995). A polymorphic record calculus and its compilation. *ACM Transactions on Programming Languages and Systems*, 17:844–895.
- Okasaki, C. (1998a). *Purely functional data structures*. Cambridge University Press.
- Okasaki, C. (1998b). Views for Standard ML. In *ACM SIGPLAN Workshop on ML*, Baltimore, Maryland.
- Okasaki, C. (1999). From fast exponentiation to square matrices: an adventure in types. In (ICFP99, 1999), pages 28–35.
- Page, R. (2003). Software is discrete mathematics. In (ICFP03, 2003), pages 79–86.
- Page, R. and Moe, B. (1993). Experience with a large scientific application in a functional language. In (FPCA93, 1993).
- Paterson, R. (2001). A new notation for arrows. In *International Conference on Functional Programming*, pages 229–240. ACM Press.
- Paterson, R. (2003). Arrows and computation. In Gibbons, J. and de Moor, O., editors, *The Fun of Programming*, pages 201–222. Palgrave.
- Paulson, L. (2004). Organizing numerical theories using axiomatic type classes. *Journal of Automated Reasoning*, 33(1):29–49.
- Perry, N. (1991a). An extended type system supporting polymorphism, abstract data types, overloading and inference. In *Proc 15th Australian Computer Science Conference*.
- Perry, N. (1991b). *The Implementation of Practical Functional Programming Languages*. Ph.D. thesis, Imperial College, London.
- Peterson, J., Hager, G., and Hudak, P. (1999a). A language for declarative robotic programming. In *International Conference on Robotics and Automation*.
- Peterson, J., Hudak, P., and Elliott, C. (1999b). Lambda in motion: Controlling robots with Haskell. In *First International Workshop on Practical Aspects of Declarative Languages*. SIGPLAN.
- Peyton Jones, S. (1987). *The Implementation of Functional Programming Languages*. Prentice Hall.
- Peyton Jones, S. (2001). Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In Hoare, C., Broy, M., and Steinbrueggen, R., editors, *Engineering Theories of Software Construction*,

- Marktoberdorf Summer School 2000*, NATO ASI Series, pages 47–96. IOS Press.
- Peyton Jones, S., Eber, J.-M., and Seward, J. (2000). Composing contracts: an adventure in financial engineering. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, pages 280–292, Montreal. ACM Press.
- Peyton Jones, S., Gordon, A., and Finne, S. (1996). Concurrent Haskell. In *23rd ACM Symposium on Principles of Programming Languages (POPL'96)*, pages 295–308, St Petersburg Beach, Florida. ACM Press.
- Peyton Jones, S., Hall, C., Hammond, K., Partain, W., and Wadler, P. (1993). The Glasgow Haskell Compiler: a technical overview. In *Proceedings of Joint Framework for Information Technology Technical Conference, Keele*, pages 249–257. DTI/SERC.
- Peyton Jones, S., Jones, M., and Meijer, E. (1997). Type classes: an exploration of the design space. In Launchbury, J., editor, *Haskell workshop*, Amsterdam.
- Peyton Jones, S. and Launchbury, J. (1991). Unboxed values as first class citizens. In Hughes, R., editor, *ACM Conference on Functional Programming and Computer Architecture (FPCA'91)*, volume 523 of *Lecture Notes in Computer Science*, pages 636–666, Boston. Springer.
- Peyton Jones, S., Reid, A., Hoare, C., Marlow, S., and Henderson, F. (1999). A semantics for imprecise exceptions. In *ACM Conference on Programming Languages Design and Implementation (PLDI'99)*, pages 25–36, Atlanta. ACM Press.
- Peyton Jones, S., Vytiniotis, D., Weirich, S., and Shields, M. (2007). Practical type inference for arbitrary-rank types. *Journal of Functional Programming*, 17:1–82.
- Peyton Jones, S. and Wadler, P. (1993). Imperative functional programming. In *20th ACM Symposium on Principles of Programming Languages (POPL'93)*, pages 71–84. ACM Press.
- Peyton Jones, S., Washburn, G., and Weirich, S. (2004). Wobbly types: type inference for generalised algebraic data types. Microsoft Research.
- Peyton Jones, S. L. (1992). Implementing lazy functional languages on stock hardware: The spineless tagless G-machine. *Journal of Functional Programming*, 2(2):127–202.
- Pierce, B. (2002). *Types and Programming Languages*. MIT Press.
- Pope, B. (2005). Declarative debugging with Buddha. In Vene, V. and Uustalu, T., editors, *Advanced Functional Programming, 5th International School, AFP 2004, Tartu, Estonia, August 14–21, 2004, Revised Lectures*, volume 3622 of *Lecture Notes in Computer Science*. Springer.
- POPL00 (2000). *27th ACM Symposium on Principles of Programming Languages (POPL'00)*, Boston. ACM.
- Pottier, F. and Régis-Gianas, Y. (2006). Stratified type inference for generalized algebraic data types. In *ACM Symposium on Principles of Programming Languages (POPL'06)*, Charleston. ACM.
- Queinnec, C. (2000). The influence of browsers on evaluators or, continuations to program web servers. In *International Conference on Functional Programming*.
- Ranta, A. (2004). Grammatical framework. *Journal of Functional Programming*, 14(2):145–189.
- Rees, J. and Clinger, W. (1986). Revised report on the algorithmic language scheme. *ACM SIGPLAN Notices*, 21:37–79.
- Rojemo, N. (1995a). *Garbage Collection and Memory Efficiency in Lazy Functional Languages*. Ph.D. thesis, Department of Computing Science, Chalmers University.
- Rojemo, N. (1995b). Highlights from nhc: a space-efficient Haskell compiler. In *(FPCA95, 1995)*.
- Roundy, D. (2005). Darcs home page. <http://www.darcs.net>.
- Runciman, C. and Wakeling, D. (1992). Heap profiling a lazy functional compiler. In (Launchbury and Sansom, 1992), pages 203–214.
- Runciman, C. and Wakeling, D. (1993). Heap profiling of lazy functional programs. *Journal of Functional Programming*, 3(2):217–246.
- Rjemo, N. and Runciman, C. (1996a). Lag, drag, void, and use: heap profiling and space-efficient compilation revisited. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'96)*, pages 34–41. ACM, Philadelphia.
- Rjemo, N. and Runciman, C. (1996b). New dimensions in heap profiling. *Journal of Functional Programming*, 6(4).
- Sage, M. (2000). FranTk: a declarative GUI language for Haskell. In *(ICFP00, 2000)*.
- Sansom, P. and Peyton Jones, S. (1995). Time and space profiling for non-strict, higher-order functional languages. In *22nd ACM Symposium on Principles of Programming Languages (POPL'95)*, pages 355–366. ACM Press.
- Schechter, G., Elliott, C., Yeung, R., and Abi-Ezzi, S. (1994). Functional 3D graphics in C++ — with an object-oriented, multiple dispatching implementation. In *Proceedings of the 1994 Eurographics Object-Oriented Graphics Workshop*. Eurographics, Springer Verlag.
- Scheevel, M. (1984). NORMA SASL manual. Technical report, Burroughs Corporation Austin Research Center.
- Scheevel, M. (1986). NORMA — a graph reduction processor. In *Proc ACM Conference on Lisp and Functional Programming*, pages 212–219.
- Scholz, E. (1998). Imperative streams – a monadic combinator library for synchronous programming. In *(ICFP98, 1998)*.
- Scott, D. (1976). Data types as lattices. *SIAM Journal on Computing*, 5(3):522–587.
- Scott, D. and Strachey, C. (1971). Towards a mathematical semantics for computer languages. PRG-6, Programming Research Group, Oxford University.
- Shapiro, E. (1983). *Algorithmic Debugging*. MIT Press.
- Sheard, T. (2004). Languages of the future. In *ACM Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA'04)*.
- Sheard, T. and Pasalic, E. (2004). Meta-programming with built-in type equality. In *Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-languages (LFM'04)*, Cork.
- Sheard, T. and Peyton Jones, S. (2002). Template meta-programming for Haskell. In Chakravarty, M., editor, *Proceedings of the 2002 Haskell Workshop, Pittsburgh*.
- Sheeran, M. (1983). *μ FP — An Algebraic VLSI Design Language*. PhD thesis, Programming Research Group, Oxford University.
- Sheeran, M. (1984). μ FP, a language for VLSI design. In *Symp. on LISP and Functional Programming*. ACM.

- Sheeran, M. (2005). Hardware design and functional programming: a perfect match. *Journal of Universal Computer Science*, 11(7):1135–1158. http://www.jucs.org/jucs_11_7/hardware_design_and_functional.
- Shields, M. and Peyton Jones, S. (2001). Object-oriented style overloading for Haskell. In *Workshop on Multi-Language Infrastructure and Interoperability (BABEL'01)*, Florence, Italy.
- Shields, M. and Peyton Jones, S. (2002). Lexically scoped type variables. Microsoft Research.
- Sinclair, D. (1992). Graphical user interfaces for Haskell. In (Launchbury and Sansom, 1992), pages 252–257.
- Singh, S. and Slous, R. (1998). Accelerating Adobe Photoshop with reconfigurable logic. In *IEEE Symposium on Field-Programmable Custom Computing Machines*. IEEE Computer Society Press.
- Somogyi, Z., Henderson, F., and Conway, T. (1996). The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*.
- Sparud, J. and Runciman, C. (1997). Tracing lazy functional computations using redex trails. In *International Symposium on Programming Languages Implementations, Logics, and Programs (PLILP'97)*, volume 1292 of *Lecture Notes in Computer Science*, pages 291–308. Springer Verlag.
- Spivey, M. and Seres, S. (2003). Combinators for logic programming. In Gibbons, J. and de Moor, O., editors, *The Fun of Programming*, pages 177–200. Palgrave.
- Steele, G. (1993). Building interpreters by composing monads. In *21st ACM Symposium on Principles of Programming Languages (POPL'94)*, pages 472–492, Charleston. ACM.
- Steele, Jr., G. L. (1978). Rabbit: A compiler for Scheme. Technical Report AI-TR-474, Artificial Intelligence Laboratory, MIT, Cambridge, MA.
- Stosberg, M. (2005). Interview with David Roundy of Darcs on source control. *OSDir News*.
- Stoye, W., Clarke, T., and Norman, A. (1984). Some practical methods for rapid combinator reduction. In (LFP84, 1984), pages 159–166.
- Strachey, C. (1964). Towards a formal semantics. In *Formal Language Description Languages for Computer Programming*, pages 198–220. North Holland. IFIP Working Conference.
- Sulzmann, M. (2003). A Haskell programmer’s guide to Chameleon. Available at <http://www.comp.nus.edu.sg/~sulzmann/chameleon/download/haskell.html>.
- Sulzmann, M. (2006). Extracting programs from type class proofs. In *International Symposium on Principles and Practice of Declarative Programming (PPDP'06)*, pages 97–108, Venice. ACM.
- Sulzmann, M., Duck, G., Peyton Jones, S., and Stuckey, P. (2007). Understanding functional dependencies via constraint handling rules. *Journal of Functional Programming*, 17:83–130.
- Sussman, G. and Steele, G. (1975). Scheme — an interpreter for extended lambda calculus. AI Memo 349, MIT.
- Swierstra, S. and Duponcheel, L. (1996). *Deterministic, Error-Correcting Combinator Parsers*, pages 184–207. Number 1129 in Lecture Notes in Computer Science. Springer Verlag, Olympia, Washington.
- Syme, D. (2005). Initialising mutually-referential abstract objects: the value recursion challenge. In Benton, N. and Leroy, X., editors, *Proc ACM Workshop on ML (ML'2005)*, pages 5–26, Tallinn, Estonia.
- Taha, W. and Sheard, T. (1997). Multi-stage programming with explicit annotations. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '97)*, volume 32 of *SIGPLAN Notices*, pages 203–217. ACM, Amsterdam.
- Tang, A. (2005). Pugs home page. <http://www.pugscode.org>.
- Tarditi, D., Morrisett, G., Cheng, P., Stone, C., Harper, R., and Lee, P. (1996). TIL: A type-directed optimizing compiler for ML. In *ACM Conference on Programming Languages Design and Implementation (PLDI'96)*, pages 181–192. ACM, Philadelphia.
- Thiemann, P. (2002a). A typed representation for HTML and XML documents in Haskell. *Journal of Functional Programming*, 12(5):435–468.
- Thiemann, P. (2002b). Wash/cgi: Server-side web scripting with sessions and typed, compositional forms. In *Practical Applications of Declarative Languages*, pages 192–208. Springer Verlag LNCS 2257.
- Turner, D. A. (1976). The SASL language manual. Technical report, University of St Andrews.
- Turner, D. A. (1979a). Another algorithm for bracket abstraction. *Journal of Symbolic Logic*, 44(2):267–270.
- Turner, D. A. (1979b). A new implementation technique for applicative languages. *Software Practice and Experience*, 9:31–49.
- Turner, D. A. (1981). The semantic elegance of applicative languages. In *Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture*, pages 85–92. ACM.
- Turner, D. A. (1982). Recursion equations as a programming language. In Darlington, J., Henderson, P., and Turner, D., editors, *Functional Programming and its Applications*. CUP.
- Turner, D. A. (1985). Miranda: A non-strict functional language with polymorphic types. In (Jouannaud, 1985), pages 1–16. This and other materials on Miranda are available at <http://miranda.org.uk>.
- Turner, D. A. (1986). An overview of Miranda. *SIGPLAN Notices*, 21(12):158–166.
- van Heijenoort, J. (1967). *From Frege to Gödel, A Sourcebook in Mathematical Logic*. Harvard University Press.
- van Rossum, G. (1995). Python reference manual. Technical Report Report CS-R9525, CWI, Amsterdam.
- Vuillemin, J. (1974). Correct and optimal placement of recursion in a simple programming language. *Journal of Computer and System Sciences*, 9.
- Wadler, P. (1985). How to replace failure by a list of successes. In (Jouannaud, 1985), pages 113–128.
- Wadler, P. (1987). Views: a way for pattern matching to cohabit with data abstraction. In *14th ACM Symposium on Principles of Programming Languages*, Munich.
- Wadler, P. (1989). Theorems for free! In MacQueen, editor, *Fourth International Conference on Functional Programming and Computer Architecture, London*. Addison Wesley.
- Wadler, P. (1990a). Comprehending monads. In *Proc ACM Conference on Lisp and Functional Programming, Nice*. ACM.

- Wadler, P. (1990b). Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248.
- Wadler, P. (1992a). Comprehending monads. *Mathematical Structures in Computer Science*, 2:461–493.
- Wadler, P. (1992b). The essence of functional programming. In *20th ACM Symposium on Principles of Programming Languages (POPL'92)*, pages 1–14. ACM, Albuquerque.
- Wadler, P. (2003). A prettier printer. In Gibbons, J. and de Moor, O., editors, *The Fun of Programming*. Palgrave.
- Wadler, P. and Blott, S. (1989). How to make ad-hoc polymorphism less ad hoc. In *Proc 16th ACM Symposium on Principles of Programming Languages, Austin, Texas*. ACM.
- Wadler, P., Taha, W., and MacQueen, D. (1988). How to add laziness to a strict language, without even being odd. In *Workshop on Standard ML, Baltimore*.
- Wadsworth, C. (1971). *Semantics and Pragmatics of the Lambda Calculus*. PhD thesis, Oxford University.
- Wallace, M. (1998). The nhc98 web pages. Available at <http://www.cs.york.ac.uk/fp/nhc98>.
- Wallace, M., Chitil, Brehm, T., and Runciman, C. (2001). Multiple-view tracing for Haskell: a new Hat. In (Haskell01, 2001).
- Wallace, M. and Runciman, C. (1998). The bits between the lambdas: binary data in a lazy functional language. In *International Symposium on Memory Management*.
- Wallace, M. and Runciman, C. (1999). Haskell and XML: Generic combinators or type-based translation. In (ICFP99, 1999), pages 148–159.
- Wan, Z. (December 2002). *Functional Reactive Programming for Real-Time Embedded Systems*. PhD thesis, Department of Computer Science, Yale University.
- Wan, Z. and Hudak, P. (2000). Functional reactive programming from first principles. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation (PLDI)*, pages 242–252, Vancouver, BC, Canada. ACM.
- Wan, Z., Taha, W., and Hudak, P. (2001). Real-time FRP. In *Proceedings of Sixth ACM SIGPLAN International Conference on Functional Programming*, Florence, Italy. ACM.
- Wan, Z., Taha, W., and Hudak, P. (2002). Event-driven FRP. In *Proceedings of Fourth International Symposium on Practical Aspects of Declarative Languages*. ACM.
- Watson, I. and Gurd, J. (1982). A practical data flow computer. *IEEE Computer*, pages 51–57.
- Wile, D. (1973). *A Generative, Nested-Sequential Basis for General Purpose Programming Languages*. PhD thesis, Dept. of Computer Science, Carnegie-Mellon University. First use of *sections*, on page 30.
- Xi, H., Chen, C., and Chen, G. (2003). Guarded recursive datatype constructors. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 224–235. ACM Press.
- Young, J. (1988). *The Semantic Analysis of Functional Programs: Theory and Practice*. PhD thesis, Yale University, Department of Computer Science.

Appendix: Content Guidelines for Authors

The criteria for the programming languages considered appropriate for HOPL-III are:

1. The programming language came into existence before 1996, that is, it was designed and described at least 11 years before HOPL-III (2007).
2. The programming language has been widely used since 1998 either (i) commercially or (ii) within a specific domain. In either case, “widely used” implies use beyond its creators.
3. There also are some research languages which had great influence on widely used languages that followed them. As long as the research language was used by more than its own inventors, these will be considered to be appropriate for discussion at HOPL-III.

Please be sure to include within your paper a clear indication of how the subject material satisfies these criteria (i.e., 1&2 or 1&3). This information can certainly be provided indirectly as part of the overall text. (For instance, some of the criteria are dates; it suffices to include the dates as part of the historical narrative.)

The main purpose of these guidelines is to help you develop the appropriate content for your contribution to HOPL-III. The questions herein point to the kind of information that people want to know about the history of programming languages. A set of questions is included for each of the major areas to be covered by HOPL-III:

- Early history of a specific language
- Later evolution of a specific language (usually a language treated in HOPL-I or in HOPL-II)

The sets of questions overlap to some extent. This guideline includes both sets of questions, in the hope that having the other set available may prove useful to you.

Even within a single set, the same question, or very similar questions, may be asked in different contexts. Please draft your paper in light of these different emphases and contexts.

Your paper should try to answer as many questions as possible in your topic area: it is understood that you might not be able to address every one of them. Because history can unfold in so many different ways, some of the questions may be clearly irrelevant to your particular topic, or your particular point of view. The information requested might no longer be available, but please remember this information is important as well. Several questions are of the form “How did something affect something else?”—it can be important for the historical record to assert that “it didn’t.”

The question set suggests the content, not the form, of your paper. (In particular, your paper should not be in question-answer format.) The questions are organized into topics and subtopics for your convenience during your research; this structure is not meant as

an outline for your paper. (Topics, subtopics, and questions are also numbered and lettered for convenience of reference.) Please feel free to use whatever form and style seems most appropriate and comfortable to you.

The Program Committee strongly suggests that you examine at least one paper from each of the proceedings of HOPL-I and HOPL-II to see what earlier contributors did. The references are:

History of Programming Languages,
Edited by, Richard L. Wexelblat,
Academic Press, 1981

History of Programming Languages-II,
Edited by, Thomas J. Bergin, Jr. and Richard G. Gibson, Jr.,
ACM Press/Addison-Wesley Publishing Company, 1996

The first two History of Programming Languages conferences established high technical and editorial standards. We trust that your contribution will help HOPL-III maintain or surpass these standards.

QUESTIONS ON THE EARLY HISTORY OF A LANGUAGE

I. BACKGROUND

1. Basic Facts about Project Organization and People

- (a) Under what organizational auspices was the language developed (e.g., name of company, department/division in the company, university?) Be as specific as possible about organizational subunits involved.
- (b) Were there problems or conflicts within the organization in getting the project started? If so, please indicate what these were and how they were resolved.
- (c) What was the source of funding (e.g., research grant, company R&D, company production units, or a government contract?)
- (d) Who were the people on the project and what was their relationship to the author(s) (e.g., team members, subordinates)? To the largest extent possible, name all the people involved, including part-timers, when each person joined the project and what each person worked on. Indicate the technical experience and background of each participant, including formal education.
- (e) Was the development done originally as a research project, as a development project, as a committee effort, as an open-source effort, as a one-person effort with some minor assistance, or....?
- (f) Was there a defined leader to the group? If so, what was his or her exact position (and title) and how did he or she get to be the leader? (e.g., appointed “from above”, self-starter, volunteer, elected?)
- (g) Was there a de facto leader different from the defined leader? If so, who was this leader and what made them the de facto leader, i.e., personality, background, experience, a “higher authority” or something else?
- (h) Were there consultants from outside the project who had a formal connection to it? If so, who were they, how and why were they chosen, and how much help were they? Were there also informal consultants? If so, please answer the same questions.
- (i) Did the participants of the project view themselves primarily as language designers, as implementers, or as eventual users? If there were some of each working on the project, indicate the split as much as possible. How did this internal view of the people involved affect the initial planning and organization of the project?
- (j) Did the language designers know (or believe) that they would also have the responsibility for implementing the first version? Whether the answer is “yes” or “no” and was the technical language design affected by this?

2. Costs and Schedules

- (a) Was there a budget? Did the budget provide a fixed upper limit on the costs? If so, how much money was to be allocated and in what ways? What external or internal factors led to the budget constraints? Was the money formally divided between language design and actual implementation? If so, indicate in what way?
- (b) Was there a fixed deadline for completion of the project? Was the project divided into phases and did these have deadlines? How well were the deadlines met?
- (c) What is the best estimate for the amount of human resources involved (i.e., in person-years)? How much was for language design, for documentation, and for implementation?
- (d) What is the best estimate of the costs prior to putting the first system in the hands of the first users? If possible, show as much breakdown on this as possible.
- (e) If there were cost and/or schedule constraints, how did that affect the language design and in what ways?

3. Basic Facts About Documentation

(a) In the planning stage, was there consideration of the need for documentation of the work as it progressed? If so, was it for internal communication among project members or external monitoring of the project by others, or both?

(b) What types of documentation were decided upon?

(c) To the largest extent possible, cite both dates and documents for the following (including internal papers and web sites which may not have been released outside of the project) by title, date, and author. (In items c1, c4, c9, and c10, indicate the level of formality of the specifications – e.g., English, formal notation – and what kind.)

* (c1) Initial idea

* (c2) First documentation of initial idea

* (c3) Preliminary specifications

* (c4) “Final” specifications (i.e., those which were intended to be implemented)

* (c5) “Prototype” running (i.e., as thoroughly debugged as the state of the art permitted, but perhaps not all of the features included)

* (c6) “Full” language compiler (or interpreter) was running

* (c7) Usage on real problems done by the developers

* (c8) Usage on real problems done by people other than the developers

- * (c9) Documentation by formal methods
- * (c10) Paper(s) in professional journals or conference proceedings
- * (c11) Please identify extensions, modifications and new versions

4. Languages and Systems Known at the Time

- (a) What specific languages were known to you and/or other members of the development group at the time the work started? Which others did any of you learn about as the work progressed? How much did you know about these languages and in what ways (e.g., as users, from having read unpublished and/or published papers, informal conversations)? (Please try to distinguish between what you, as the writer knew and what the other members of the project knew.)
- (b) Were these languages considered as formal inputs that you were definitely supposed to consider in your own language development, or did they merely provide background? What was it about these languages that you wanted to emulate (e.g., syntax, capabilities, internal structure, application area, etc.)?
- (c) How influenced were you by these languages? Put another way, how much did the prior language backgrounds of you and other members of the group influence the early language design? Whether the answer is “a lot” or “a little,” why did these other languages have that level of influence? (This point may be more easily considered in Section II: Rationale of Content of Language.)
- (d) Was there a primary source of inspiration for the language and if so, what was it? Was the language modeled after this (or any other predecessors or prototypes)?

5. Intended Purposes and Users

- (a) For what application area was the language designed, i.e., what type of problems was it suppose to be used for? Be as specific as possible in describing the application area; for example, was “business data processing” or “scientific applications” ever carefully defined? Was the apparent application area of some other language used as a model?
- (b) For what types of users was the language intended (e.g., experienced programmers, mathematicians, business people, novice programmers, non-programmers)? Was there any conflict within the group on this? Were compromises made, and if so, were they made for technical or non-technical reasons?
- (c) What equipment was the language intended to be implemented on? Wherever possible, cite specific machine(s) by manufacturer(s) and number, or alternatively, give the broad descriptions of the time period with examples (e.g., “COBOL was defined to be used on ‘large’ machines which at that time included UNIVAC I and II, IBM 705.”) Was

machine independence a significant design goal, albeit within this class of machines?
(See also Question (1b) in Rationale of the Content of the Language.)

6. Source of Motivation

- (a) What (or who) was the real origin of the idea to develop this language?
- (b) What was the primary motivation in developing the language (e.g., research, task assigned by management?)

II.RATIONALE OF THE CONTENT OF THE LANGUAGE

These questions are intended to stimulate thought about various factors that affect most language design effort. Not all the questions are relevant for every language. They are intended to suggest areas that might be addressed in each paper.

1. Environment Factors

To what extent was the design of the language influenced by:

- (a) Program size: Was it explicitly thought that programs written in the language would be large and/or written by more than one programmer? What features were explicitly included (or excluded) for this reason? If this factor wasn't considered, did it turn out to be a mistake? Were specific tools or development environments designed at the same time to support these choices?
- (b) Program libraries: Were program libraries envisioned as necessary or desirable, and if so, how much provision was made for them?
- (c) Portability: How important was the goal of machine independence? What features reflect concern for portability? How well was this goal attained? See also question (1) on Standardization and question (5c) under Background.
- (d) User Background and Training: What features catered to the expected background of intended users? In retrospect, what features of the language proved to be difficult for programmers to use correctly? Did some features fall into disuse? Please identify such features and explain why they fell into disuse? How difficult did it prove to train users in the correct and effective use of the language, and was the difficulty a surprise? What changes in the language would have alleviated training problems? Were any proposed features rejected because it was felt users would not be able to use them correctly or appropriately?
- (e) Execution Efficiency: How did requirements for executable code size and speed affect the language design? Were programs in the language expected to execute on large or small computers (i.e., was the size of object programs expected to pose a problem)? What design decisions were explicitly motivated by the concern (or lack of concern) for execution efficiency? Did these concerns turn out to be accurate? How was the design of specific features changed to make it easier to optimize executable code?
- (f) Target Computer Architecture: To what extent were features in the language dictated by the anticipated target computer, e.g., its word size, existence of floating-point hardware, instruction set peculiarities, availability and use of index registers, special-purpose co-processors and accelerators, etc.?
- (g) Compilation Environment: To what extent, if any, did concerns about compilation efficiency affect the design? Were features rejected or included primarily to make it easier to implement compilers for the language or to ensure that the compiler(s) would

execute quickly? In retrospect, how correct or incorrect do you feel these decisions were? What decisions did you make regarding use of the compiler run-time system?

(h) Programming Ease: To what extent was the ease of coding an important consideration and what features in the language reflect the relative importance of this goal? Did maintainability considerations affect any design decisions? If so, which ones?

(i) Execution Environment: To what extent did the language design reflect its anticipated use in a batch, embedded, portable, office, or networked environment? What features reflect these concerns?

(j) Parallel Implementation: Were there multiple implementations being developed at the same time as the later part of the language development? If so, was the language design hampered or influenced by this in any way?

(k) Standardization: In addition to (or possibly separate from) the issue of portability, what considerations were given to possible standardization? What types of standardization were considered, and what groups were involved and when?

(l) Networking/Parallel Environment: To what extent did the language design reflect its anticipated use in a networked- or parallel-execution environment? What features reflect these concerns?

2. Functions to be Programmed

(a) How did the operations and data types in the language support the writing of particular kinds of algorithms?

(b) What features might have been left out, if a slightly different application area has been in mind?

(c) What features were considered essential to properly express the kinds of programs to be written?

(d) What misconceptions about application requirements turned up that necessitated redesign of these application specific features before the language was actually released?

3. Language Design Principles

(a) What consideration, if any, was given to designing the language so that programming errors could be detected early and easily? Were the problems of debugging and testing considered? Were debugging and testing facilities deliberately included in the language?

(b) To what extent was the goal of keeping the language simple considered important? What kind of simplicity was considered most important? What did your group mean by “simplicity”?

(c) What thought was given to make programs more understandable and how did these considerations influence the design? Was there conscious consideration of making programs “easy to read” versus “easy to write”? If so, which were chosen and why?

(d) Did you consciously develop the data types first and then the operations, or did you use the opposite order, or did you try to develop both in parallel with appropriate iteration? Were data and operations combined into objects?

(e) To what extent did the design reflect a conscious philosophy of how languages should be designed (or how programs should be developed)? What was this philosophy?

4. Language Definition

(a) What techniques for defining languages were known to you? Did you use these or modify them, or did you develop new ones?

(b) To what extent--if any-- was the language itself influenced by the technique used for the definition?

5. Concepts About Other Languages

(a) Were you consciously trying to introduce new concepts? If so, what were they? Do you feel that you succeeded?

(b) If you were not trying to introduce new concepts, what was the justification for introducing this new language? (Such justification might involve technical, political, or economic factors.)

(c) To what extent did the design consciously borrow from previous language designs or attempt to correct perceived mistakes in other languages?

6. Influence of Non-technical Factors

(a) How did time and cost constraints (as described in the Background section) influence the technical design?

(b) How did the size and structure of the design group affect the technical design?

(c) Provide any other information you have pertaining to ways in which the technical language design was influenced or affected by non-technical factors.

III. A POSTERIORI EVALUATION

1. Meeting of Objectives

- (a) How well do you think the language met its original objectives?
- (b) Do the users think the language has met its objectives?
- (c) How well do you think the computing community (as a whole) thinks the objectives were met?
- (d) How much impact did portability (i.e., machine independence) have on the acceptance by users?
- (e) Did the objectives change over time? If so, how, when, and in what ways did they change? See also question (2d) under Rationale of Content of Language and answer here if appropriate.

2. Contributions of Language

- (a) What is the major contribution made by this language? Was this one of the objectives? Was this contribution a technical or a non-technical contribution, or both? What other important contributions are made by this language? Were these part of the define objectives? Were these contributions technical or non-technical?
- (b) What do you consider the best points of the language, even if they are not considered to be a contribution to the field (i.e., what are you proudest of, regardless of what anybody else thinks)?
- (c) How many other people or groups decided to implement this language because of its inherent value?
- (d) Did this language have any effect on the development of later hardware?
- (e) Did this language spawn any “dialects”? If so, please identify them. Were they major or minor changes to the language definition? How significant did the dialects themselves become?
- (f) In what way do you feel the computer field is better off (or worse) for having this language?
- (g) What fundamental effects on the future of language design resulted from this language development (e.g., theoretical discoveries, new data types, new control structures)?

3. Mistakes or Desired Changes

- (a) What mistakes do you think were made in the design of the language? Were any of these able to be corrected in a later version of the language? If you feel several mistakes were made, list as many as possible with some indication of the severity of each.
- (b) Even if not considered mistakes, what changes would you make if you could do it all over again?
- (c) What have been the biggest changes made to the language (albeit probably by other people) since its early development? Were these changes or new capabilities considered originally and dropped in the initial development, or were they truly later thoughts?
- (d) Have changes been suggested but not adopted? If so, be as explicit as possible about changes suggested, and why they were not adopted.

4. Problems

- (a) What were the biggest problems you had during the language design process? Did these affect the end result significantly?
- (b) What are the biggest problems the users have had?
- (c) What are the biggest problems the implementers have had? Were these deliberate, in the sense that a conscious decision was made to do something in the language design, even if it made the implementation more difficult?
- (d) What trade-offs did you consciously make during the language design process? What trade-offs did you unconsciously make?
- (e) What compromises did you have to make to meet other constraints such as time, budget, user demands, political, or other factors?

IV. IMPLICATIONS FOR CURRENT AND FUTURE LANGUAGES

1. Direct Influence

- (a) What language developments of today and the foreseeable future are being directly influenced by your language? Regardless of whether your answer is “none” or “many, such as...,” please indicate the reasons.
- (b) Is there anything in the experience of your language development which should influence current and future languages? If so, what is it? Put another way, in light of your experience, do you have advice for current and future language designers?
- (c) Does your language have a long-range future? Regardless of whether your answer is “yes” or “no”, please indicate the reasons.

2. Indirect Influence

- (a) Are there indirect influences which your language is having now? Are there any indirect influences that it can be expected to have in the near future? What are these, and why do you think they will be influential?

QUESTIONS ON THE EVOLUTION OF A PROGRAMMING LANGUAGE

The principle objective of a contribution in this category is to treat the history of a major language subsequent to its original development. In many cases, this entails extending the history of some language whose origins were treated in HOPL-I, or in HOPL-II. (Authors of papers in this category will be working with a member of the Program Committee, who will keep you informed of other relevant papers under consideration.)

When a programming language is first developed, it is typically the work of an individual or a small, concentrated group. Later development of the language is often the result of an expanded, re-staffed group, and perhaps additional individuals or groups outside the original organization. Similarly, while the original work is often focused on language design and implementation for a single environment, later developments are undertaken in a broader arena.

When compared with questions about the early history of a language, the following questions reflect this change in context. In particular, these questions are about the set of diverse development activities that surround the language, such as standardization, new implementations, significant publications, language-oriented groups (e.g., SIGs, user groups), etc.

These questions are grouped into the same four broad categories that apply to papers on the origins of a language:

- * Background
- * Rationale
- * A posteriori evaluation
- * Implications for current and future languages

The first question applies broadly to the language itself. It is intended to identify the particular development activities that are the focus of the history paper. In contrast, you should address the remaining questions for each of the activities identified in that initial background section.

You might also have significant information to contribute in response to questions raised about the early history of the language. Please examine the questions provided for authors of “early history” papers.

Where appropriate, your contribution should make reference to related papers from HOPL-I, HOPL-II, or HOPL-III.

I. BACKGROUND

1. Basic Facts About Activities, Organizations and People

(a) What are the categories of development to be discussed (e.g., standardization, new implementations, open source, significant publications) and what specific activities are reported on?

(From this point on, each question is intended to apply independently to each development activity identified in question I.1(a) above.)

(b) What organizations played principal roles in these developments? Identify them as precisely as possible: corporation and division, university and department, agency and office, etc. How were these organizations sponsored and funded?

(c) What, if any, was the nature of the cooperation or competition among these organizations?

(d) Who were the people involved in these developments? How were they related organizationally to each other and to the original developers for this language? Please be as specific as possible regarding names, titles, and dates.

(e) How did the roles of various individuals change during the course of the activity?

2. Costs and Schedules

(a) What was the source and amount of funding for supporting the development? Was it adequate?

(b) What was the schedule, if any?

(c) What was the estimated human effort required to carry it out?

(d) What was the estimated cost of the development?

(e) What were the effects of cost and schedule constraints?

3. Basic Facts About Documentation

(a) What are the significant publications arising from development? For each provide:

- A specific reference
- Names of authors (if not part of the reference)
- Intended audience (e.g., user, implementer)
- Format (e.g., manual, general trade book, standards document)
- Availability

4. Languages/Systems Known at the Time

- (a) What languages or systems other than the one in question had an effect on the development? In what ways did they affect the development?
- (b) How did information about each of these languages or systems become available to the people it influenced?

5. Intended Purposes and Users

- (a) What was the intended purpose of the development?
- (b) Were the results proprietary, for sale, freely distributed, etc.?
- (c) For whom were its results intended? How did this group of people differ from the originally intended set of users for this language?

6. Motivation

- (a) Who was the prime mover for the development?
- (b) What was the underlying motivation for the development?

II. RATIONALE FOR THE CONTENT OF THE DEVELOPMENT

To the extent that is appropriate, apply the “early history” questions in each of the following subcategories to the activity being addressed.

1. Environment Factors

- (a) What were the effects on the development of concerns about program size, program libraries, portability, user background, execution efficiency, target computer architecture and speed, compilation environment, programming ease, execution environment, character set, parallel implementation, standardization, networked or parallel environment?
- (b) In what ways had the environment changed since the original development of the language?

2. Expected Applications of the Language

- (a) How did expected applications influence choice of operations, data types, and objects?
- (b) What features were essential to meet intended applications?

3. Design Principles Applied to the Development

- (a) What, if any, was the underlying, consciously applied design philosophy?
- (b) What considerations were made for detecting and correcting errors in the development?
- (c) What role did “simplicity” play, and what was meant by “simplicity”?
- (d) What role did “understandability” play? Which was given higher priority: “ease of reading” or “ease of writing”, and why?
- (e) Were certain aspects of language (e.g., data types, operations, objects) considered more fundamental to the development than others? Why?

4. Language Definition

- (a) What language definition techniques were used in this development? To what extent was the result of the development influenced by these choices?

5. Concepts of Other Languages

(a) To what extent was the introduction of new language concepts or features a part of this development?

(b) In what ways did concepts from other languages influence this development?

6. Influence of Non-Technical Factors

(a) What was the effect of other, similar developments on this one (e.g., overlapping standardization efforts)?

(b) What was the effect of time and cost constraints on the development?

(c) How did the size and structure of the development group affect results?

III. A POSTERIORI EVALUATION

1. Meeting of Objectives

- (a) How well did the development meet its objectives?
- (b) How well did the users feel the development met its objectives?
- (c) What was the reaction of the computing community at large?
- (d) How did portability of results impact their acceptance?
- (e) Did the objectives of the development change over time? If so, when, how, and why did they change?

2. Contributions of the Development

- (a) What were the biggest contributions of this development? Were they among the original objectives?
- (b) What do you consider its best features? What do you consider its worst features?
- (c) How has this development affected other activities (e.g., development of other languages, dialects, language processors, standards, operating systems, and computer hardware)? Which of these other activities have become significant in their own right?
- (d) In what way is the computer field better or worse off because of this development?
- (e) What fundamental effects on programming language methodology have arisen from this development? (e.g., new data types, control structures, techniques for definition, for types of documentation, application design strategies, theoretical discoveries, etc.)

3. Mistakes or Desired Changes

- (a) What mistakes were made in the development? Were these mistakes corrected in later developments?
- (b) What changes would you now make, if you could?
- (c) What were the biggest changes made to the development results since they were first released? Had they been considered earlier and then dropped, or were they truly later thoughts?
- (d) What significant changes have been suggested but not adopted, and why?

4. Problems

- (a) What were the major obstacles in carrying out the development?
- (b) What were the major problems encountered by people who used its results: e.g., language users, designers, implementors, standards committees, etc.?
- (c) What trade-offs were made during the development? Which were made consciously and which were recognized after the fact?
- (d) What compromises were made to meet other constraints such as time, budget, user demand, and political factors?
- (e) Which estimates (time, cost, effort, and human resources) were farthest from reality? Why were they off?
- (f) What application-specific features might better have been left out?

IV. IMPLICATIONS FOR CURRENT AND FUTURE LANGUAGES

- (a) Which current and foreseeable developments are being directly or indirectly influenced by this development, and why?
- (b) Which results, if any, of this development have a long range future? Why?