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 evolution 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.

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 'revised^5' 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 Hassle!

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.

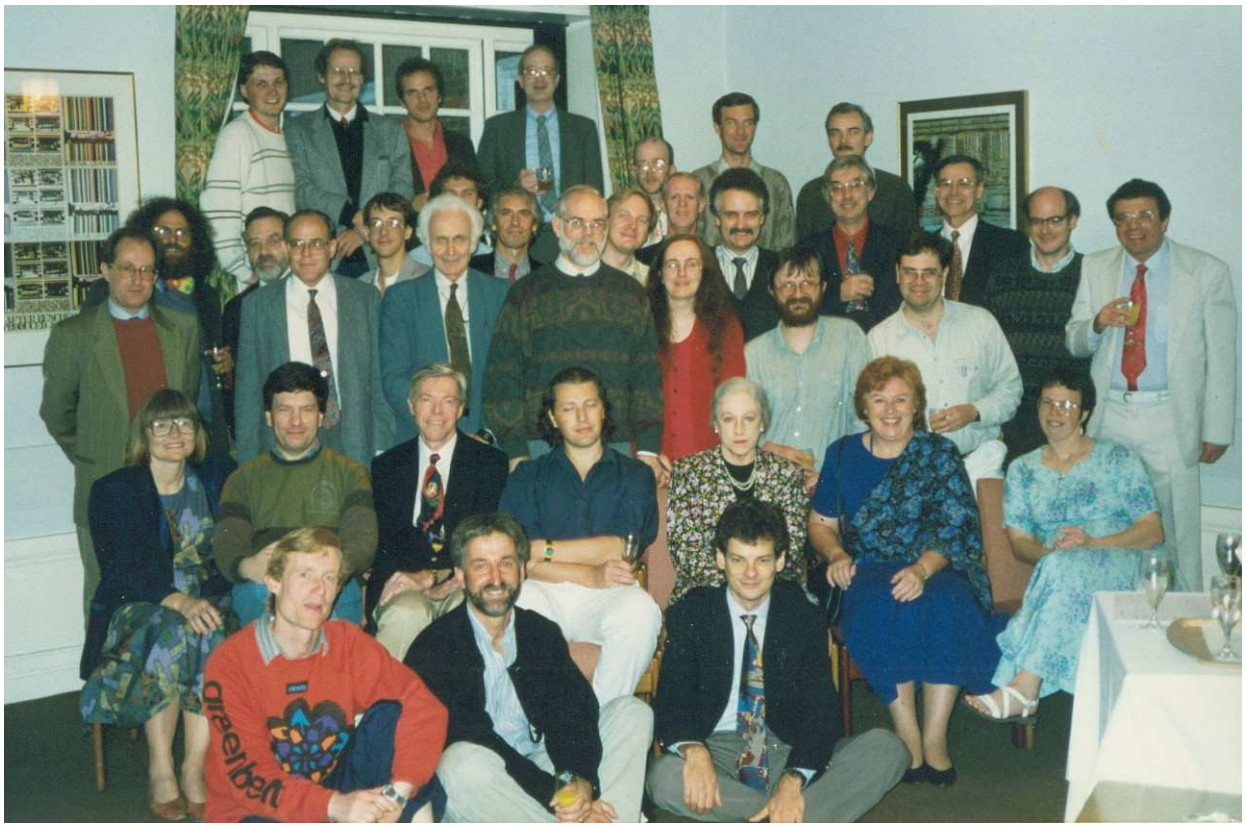
However, as Haskell became more widely used (partly because of the existence of the Haskell 98 standard), many small flaws emerged in the language design, and many ambiguities in the Report were discovered. Peyton Jones's role evolved to that of Benign Dictator of Linguistic Minutiae.

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 form[1]. Many of the implementations, libraries, and tools mentioned in the figure are discussed later in the paper.

[1] This 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 Kieburtz, 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.

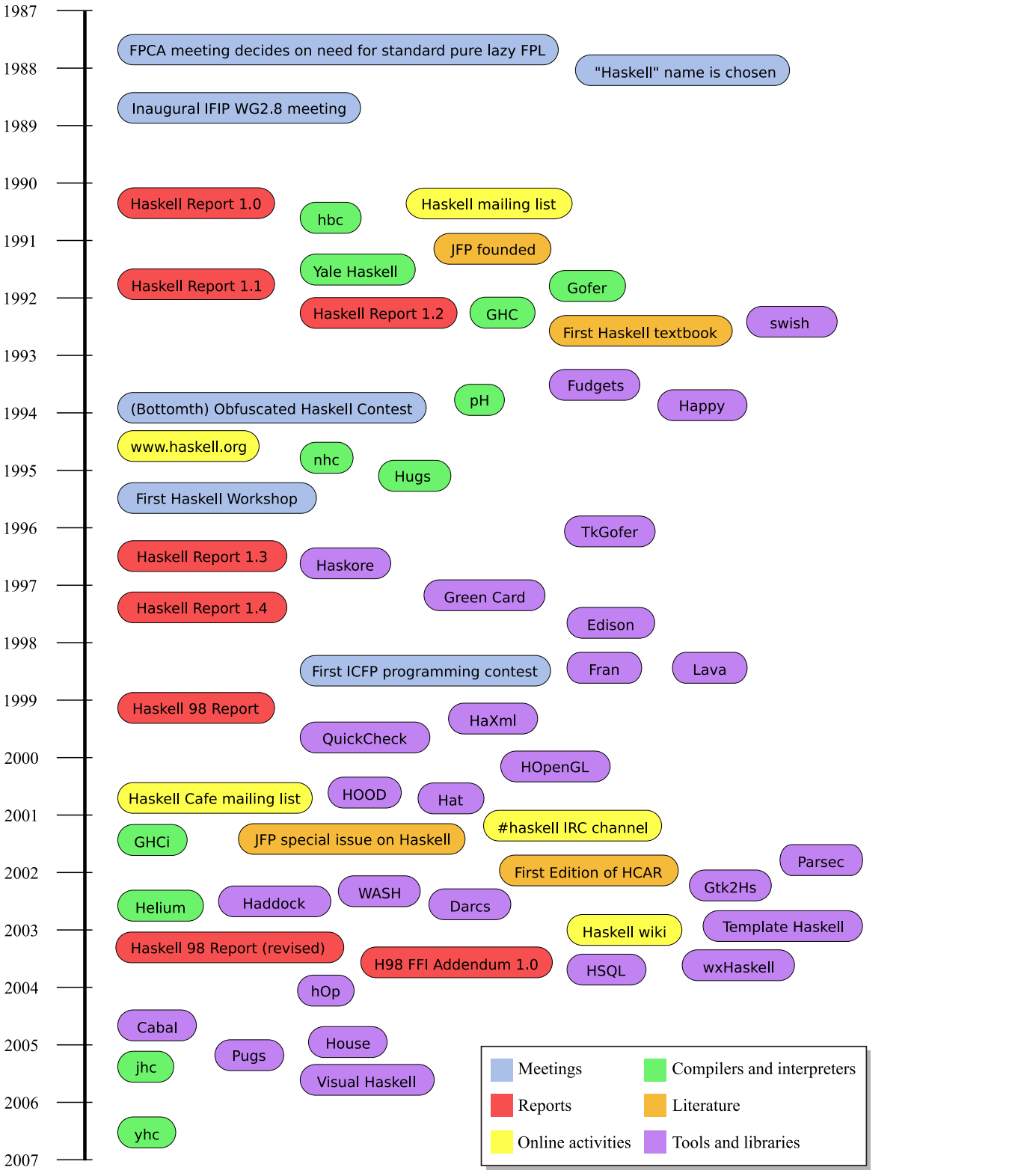


Figure 2. Haskell timeline

"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 `''a` 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 them[2].

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, `\x -> exp`, was chosen to resemble lambda expressions, since the backslash was the closest single ASCII character to the Greek letter λ. However, "`->`" 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.

[2] The same is true of Miranda users.

[3] This 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.

(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 styles[4]:

```
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.

[4] The 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 xps  = Nothing
              | otherwise = Just xp
              where
                xps = filter p xs
                xp  = head xps
```

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 nontermination 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` $(\perp, \perp)$ = `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` $\perp$ = `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 "`~`" 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 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 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 kind[5] `*->*`, 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 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.

[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.

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 $\forall a.a \rightarrow (a \rightarrow \texttt{Int}) \rightarrow \texttt{T}$; 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 $\tau \rightarrow \texttt{Int}$. 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\l)` is the offset in `r` at which a field labelled `l` 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 Sundaresh wrote a report describing them, comparing their expressiveness, and giving translations between them during these deliberations (Hudak and Sundaresh, 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 Sundaresh, 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 resps = []

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:

$$\texttt{return}\ x\ \texttt{>>=}\ f \quad = \quad f\ x$$

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 tranformer 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  :: IORef a -> IO a
writeIORef :: IORef 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 (IORef a)
readIORef  :: IORef a -> IO a
writeIORef :: IORef 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 programming[6]. 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, Herlilhy 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" status[7]. 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 packages[8]. 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 Hugs[9]

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 Loosemore, 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 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 defunctionalised 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 Formiga[10].

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 clausification 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). Combinations 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 clausify 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:

$$\texttt{seq}\ x\ y = \begin{cases} \bot, & \text{if } x = \bot \\ y, & \text{otherwise} \end{cases}$$

- 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 `\x -> ` $\bot$ from $\bot$ (since `seq` $\bot$ `0` goes into a loop, while `seq (\x -> ` $\bot$`) 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 `∀a,b.a -> b -> 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 parametrictity 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

$$\texttt{foldr}\ \bot\ 0\ (\texttt{build seq}) \quad \neq \quad \texttt{seq}\ \bot\ 0$$

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 users[11]. 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

[11] In 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 account[12] 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.

[12] See `http://www.kuro5hin.org/story/2001/7/31/0102/11014`.

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 Haddock 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`

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

$$float \quad ::= \quad sign^{?}\ digit^{+}\ ('.'\ digit^{+})^{?}$$

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 Elliot 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 μFP (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 WxHaskell (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 processing[13]

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.

[13] This section is based on material contributed by Paul Callaghan.

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.

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 year[14]. 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 respondees 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 Connections[15]

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

[15] This 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 Bluespec[16]

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.

[16] This section was contributed by Rishiyur Nikhil of Bluespec.

12.4.3 Aetion[17]

Aetion 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 Aetion 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 Aetion 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. Aetion 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 Aetion's rare use of such an agreeable programming language promotes employee retention.

The main difficulty that Aetion 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 Aetion 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 Aetion 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 Aetion is no longer willing or able to do so.)

12.4.4 Linspire[18]

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

[17] This section was contributed by Mark Carroll of Aetion.

[18] This section was contributed by Clifford Beshers of Linspire.

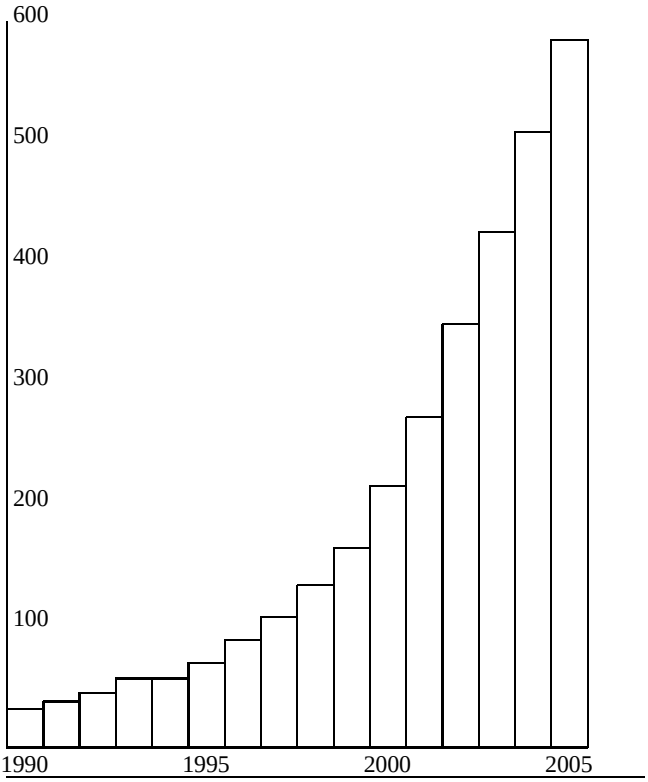


Figure 7. Growth of the "hard-core" Haskell community

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 Report[19], 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 August[21]. 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 bloggers[23], 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 funded[24].

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 respondees 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 respondees when they first learnt Haskell, so we could estimate how the size of the community has changed over the years[25]. The results are shown in Figure 7, where the bars show the total number of respondees 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 programming[26]!

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 respondees 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 programming[27]. 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: the UK is in fourth place (49), and Sweden in sixth (29). Other countries with 20 or more respondees 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

[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.

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 lighthearted sign of that is Haskell's ranking in the Great Computer Language Shootout[28]. 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.

[28] See `http://shootout.alioth.debian.org`

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 Kieburtz (Oregon Graduate

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: Thiaggo 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 Voigtländer, Johannes Waldmann, Malcolm Wallace, Mitchell Wand, Eric Williger, and Marc van Woerkom.

Some sections of this paper are based directly on material contributed by Lennart Augustsson, Clifford Beshers, 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 pemutation 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 multithreaded 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 Tech-

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. 1*. 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)*, Cophenhagen. 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, Cophenhagen. 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 Univerity, 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*, 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 Voigtländer, 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 Univesrity.

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). Revised^5 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., Micheloud, 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 Applicatioons (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). *μFP — An Algebraic VLSI Design Language*. PhD thesis, Programming Research Group, Oxford University.

Sheeran, M. (1984). μFP, 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 intefaces 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 Godel, 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.