A dissertation for the Ph.D. degree in Computing Science

Reactive Objects and Functional Programming

Johan Nordlander

CHALMERS | GÖTEBORG UNIVERSITY

Department of Computing Science
Chalmers University of Technology
SE-412 96 Göteborg, Sweden

Göteborg, May 1999

A dissertation for the Ph.D. Degree in
Computing Science at Chalmers University of Technology

Department of Computing Science
Chalmers University of Technology
SE-412 96 Göteborg, Sweden

Reactive Objects and Functional Programming
Johan Nordlander
Göteborg 1999
ISBN 91-7197-823-2

Doktorsavhandlingar vid
Chalmers Tekniska Högskola
Ny serie 1521
ISSN 0346-718X

Abstract

The construction of robust distributed and interactive software is still a challenging task, despite the recent popularity-increase for languages that take advanced programming concepts into the mainstream. Several problematic areas can be identified: most languages require the *reactivity* of a system to be manually upheld by careful avoidance of blocking operations; mathematical *values* often need to be encoded in terms of stateful *objects* or vice versa; *concurrency* is particularly tricky in conjunction with encapsulated software components; and static type safety is often compromised because of the lack of simultaneous support for both *subtyping* and *polymorphism*.

This thesis presents a programming language, *O'Haskell*, that has been consciously designed with these considerations in mind. O'Haskell is defined by conservatively extending the purely functional language *Haskell* with the following features:

- A central structuring mechanism based on *reactive objects*, which unify the notions of objects and concurrent processes. Reactive objects are asynchronous, state-encapsulating servers whose purpose is to react to input messages; they cannot actively block execution or selectively filter their sources of input.

- A monadic layer of object-based computational effects, which clearly separates stateful objects from stateless values. Apart from higher-order functions and recursive data structures, the latter notion also includes first-class commands, object templates, and methods.

- A safe, polymorphic type system with declared record and datatype subtyping, supported by a powerful partial type inference algorithm.

It is claimed that these features make O'Haskell especially well-adapted for the task of modern software construction.

The thesis presents O'Haskell from both a practical and a theoretical perspective. The practical contributions are a full implementation of the language, a number of non-trivial programming examples that illustrate the merits of reactive objects in a realistic context, and several reusable programming methods for such applications as graphical user interfaces, embedded controllers, and network protocols. The theoretical results serve to substantiate the informal claim made regarding usability, and include a provably sound polymorphic subtyping system, soundness and partial completeness for the inference algorithm, a formal dynamic semantics, and a result that characterizes the conservative extension of a purely functional language with state and concurrency.

Keywords: language design, functional programming, concurrency, reactive objects, polymorphic subtyping.

Creativity is allowing yourself to make mistakes. Art is to know which ones to keep.

— Scott Adams, creator of *Dilbert*.

Acknowledgements

Ten years ago Björn von Sydow, even then a renowned lecturer, and I, his undergraduate student, were having a discussion concerning the ideal programming language during a lunch-break at the Luleå Technical University. Although the discussion was conducted in rather vague terms, I still remember getting a very clear picture of what this ideal language should look like. When I now find myself in a position where I can sum up the many years of programming language research that have ultimately led to a finished thesis, I can only conclude that the result was there all the time! *This was the language we were talking about*, and it only makes me proud to give credit where credit is due — to my supervisor Björn von Sydow, who put me on the right track long before I knew I wanted to be a research student, and who has kept me on that track ever since. Thanks, Björn, for being such an enormous source of inspiration, for believing in me, and for letting me make the right mistakes!

The Department of Computing Science at Chalmers has provided a very rich and stimulating research environment, and I would like to express my gratitude to its skilled administrative staff, and to all past and present colleagues with whom I have had so many joyful and educative encounters.

I am particularly indebted to two very special friends and colleagues. Magnus Carlsson showed an early interest in my work (despite my ranting!), and soon became involved in it himself. Chapter 6 is based on work jointly performed with him. Magnus has also provided me with numerous comments on drafts of this thesis. Thomas Hallgren and I have had long discussions on type systems and almost every other subject as well, and he also provided the first full implementation of my subtype inference algorithm. Both Magnus and Thomas have moreover been an important part of my social life in Göteborg, and the only sensible way of acknowledging their influence on the making of this thesis is by saying: thanks for everything, brothers! It has been a pleasure!

Thanks are also due to my supervision committee — Bengt Nordström, Dave Sands, and Thomas Johnson — for giving me useful comments on an earlier draft of the thesis and the work that has preceded it. The dissertation has also benefited from stimulating discussions with Lars Pareto, Lennart Augustsson, Simon Peyton Jones, Norman Ramsey, Rogardt Heldal, and Dan Synek, among others. I am moreover indebted to Fritz Henglein for patiently introducing me to the field of polymorphic subtyping, and to Philippas Tsigas and Marina Papatriantafilou for helping me sorting out the mysteries of self-stabilization.

The commuting life I have been living during this period would not have been possible without the support of all good friends in Göteborg who have opened up their homes for me. To Johanna & Sven, Gunnel & Johan, Rauni & Pontus, Björn & Vero, Patrik & Lina, Janne & Camilla: my hearty thanks for the hospitality and the good company, and I hope you all feel that my family and I are always keeping our door open for you as well.

Vivi-Anne Lundström's skilled eyes have helped me identify many weak spots in my English presentation, which I gratefully acknowledge. Vivi-Anne and Jack have also been a dependable source of encouragement over the years, as well as providers of invaluable help in numerous practical matters (especially during this last hectic period), and for that I am particularly grateful.

A very special place in my heart is reserved for Siv Lundström and Owe Wikström, whom I count as my best friends, in the true meaning of the word. This work would not have been thinkable without their care and support, and I dedicate my thesis to them with pride.

Finally, my most intimate thank you goes to Eva, my love and fellow-traveller in life. Much has changed since that distant day I left for Luleå to study and live in a caravan on a camp closed for the winter, but her love and unbroken faith in my capabilities has been a constant factor in my life. Now we share our home with three wonderful children, and to them I just want to say now that the "fairy-tale book" is finally written: *Agnes, Charlott och Oskar — nu ska vi leka!*

This dissertation is partly based on previously published material. Chapter 5 is an extended version of [Nor98], and chapter 6 is based on the work reported in [NC97]. Some original thoughts underlying the thesis can also be found in [Nor95], although the technical material contained in that report should be considered superseded by [NC97] and the present work.

Contents

Chapter 1

Introduction

Developers of modern software are faced with expectations on a much higher level than what was the case merely ten years ago. Today graphical user interfaces with multiple input sources are standard, and interactivity has become the rule in almost every computerized task. Networking, distributed computing, and mobility of software components are already commonplace, as is the embedding of computers inside highly specialized products where the computing environment looks radically different from the traditional array of files, screen, and keyboard. And even if relatively few of these systems are governed by what is called hard real-time constraints, modern software in general is nevertheless expected to present a very high degree of responsiveness, e.g. in terms of immediate confirmation of user commands, the ability to abort any command, continuous feedback of internal progress, etc. All in all, today's requirements on computer systems make programming these systems a particularly challenging task.

This dissertation is devoted to the study of programming language features that would make it easier to meet this challenge.

1.1 Background

Recent years have shown two very interesting examples of language definitions with pretensions of being specifically well-suited to modern software construction: *Erlang* in the field of telecommunications [AVWW96], and *Java* as the programming language incarnation of the Internet-boom [GJS96]. Both these designs originate from real-world demands and have very practically oriented goals, yet they incorporate several features that have so far only been seen in relatively isolated languages used mainly for teaching and research. A common example in both Erlang and Java is the use of automatic memory management with garbage collection. To this Erlang adds message-based concurrency and a declarative programming style based on recursion and pattern-matching, while Java has adopted static type safety with subtyping, and a notion of machine-

independent mobility of code.

However, although Erlang and Java represent significant advances over more traditional languages aimed at large scale program development, programming robust interactive or distributed systems in these languages still remains a challenge. Several problematic areas can be identified.

Reactivity Both Erlang and Java take on the traditional view that indefinite blocking of execution is an operational property that may be assigned to functions/methods without any specific notice in their calling interfaces. This makes the liveness, or *reactivity* of a system highly dependent on the programmer's ability to manually assert that every process, in every state, is capable of handling all input events it can possibly receive. Utmost care must be taken to ensure that no subpart of a process involves inadvertently blocking operations, since neither modules nor any other language construct can be used to detect violations of such assumptions. Indeed, one of the primary objectives behind the various design patterns and software toolkits that accompany Erlang and Java is a desire to guide the programmer towards a system structure that is likely to reduce the complexity of reactive programming to some extent.

Declarative programming Despite its advanced memory-management and its lack of primitive pointers, Java is still in essence an imperative language. Much has been said about the shortcomings of imperative languages vs. the merits of a more declarative programming style, and the arguments need not be repeated here (see e.g. [Bac78, Tur81, Hug90, Wad92] for a thorough discussion). However, modern developments have perhaps made the case for declarative programming even more relevant. Firstly, compiler technology has now advanced to a state where the efficiency of declarative languages indeed can match imperative implementations [SF98]. Secondly, the discovery of *monads* has effectively removed the previous misconception that a declarative style must be incompatible with the use of state, concurrency, exceptions, and other kinds of computational effects [Wad92, Wad97]. And thirdly, the current interest in communicating software components has emphasized the benefits of a semantic separation between stateful and stateless data structures, since stateless data may be safely communicated by reference without the need for conservative copying.

Erlang promotes a declarative style of programming that definitely has an edge over Java in this respect. Unfortunately, since Erlang intertwines operations for process communication with expression evaluation, the language loses a very important declarative property: the ability to freely replace a variable with its definition (sometimes called the *referential transparency* property). This is one area where monads are able to offer a more intuitive as well as semantically more appealing solution.

Objects and concurrency While concurrent objects appear quite naturally in informal modelings of computer software, their realization in Java brings up

technicalities of an unexpected complexity – to the point where Java toolkit designers have been forced to assume that concurrency will not be an issue for the everyday Java programmer [Gea98]. In part this amounts to Java's rather poor concurrency support, which separates threads from the notion of objects, and provides only shared memory and primitive locks for communication and synchronization. But already in the sequential case, the *inheritance* mechanism is arguably the single most complicated feature of object-oriented languages like Java. Adding concurrency to a language with inheritance multiplies this complexity, and emphasizes the conflict between encapsulation and extensibility that lies inherent in the object-oriented paradigm [Pap94, MY93, Ame87, Sny86]. The lack of satisfying solutions to this inheritance anomaly is a probable cause behind the design choice to relegate concurrency to an optional, low-level feature in Java.

In contrast, the message-based concurrency model of Erlang enables a rather straightforward implementation of the kind of encapsulated, concurrent software components that Java lacks. Inheritance is not provided, but similar mechanisms for software reuse can be encoded using higher-order functions. However, since Erlang lacks any notion of basic object-oriented features like methods and classes as well, the amount of encoding required for object-oriented programming in Erlang can be quite substantial. Worse, though, is that the absence of a type system also hinders exploration of the object-oriented classification paradigm at the level of types.

Static type safety Static detection of type errors has proven to be invaluable in the construction of robust software, and Erlang's lack of an integrated type discipline is therefore a serious shortcoming. On the other hand, the flexibility offered by an untyped language is not easily matched. Two conservative approximations are in widespread use: *subtyping* for the cases when a data value may belong to more than one type, and *parametric polymorphism* when code fragments can be uniformly defined for any choice of types.

Java offers statically safe subtyping as one of its primary features, but forces the programmer into frequent use of *type casts* in order to circumvent the lack of polymorphism. Although dynamically checked, these casts provide essentially no more robustness at run-time than untyped Erlang code does. Strongly typed functional languages like Haskell and Standard ML make heavy use of parametric polymophism [Pet97, MTH90], but their lack of subtyping often leads to unnecessarily complicated code, especially in conjunction with records.

Unfortunately, successful integrations of polymorphism and subtyping are still rare, even counting recent proposals on how to extend Java with polymorphic features [OW97, BOSW98, MBL97]. One major obstacle is that explicitly typed polymorphic code is tedious to write, and algorithms for automatic subtype-inference have yet to show practically applicable results in terms of efficiency and size of the inferred types. Still, for many modern programming tasks, a polymorphic subtyping system would make a most powerful addition, not least for expressing the communication contracts that must exist between

the autonomous components of a concurrent, distributed, and interactive system.

The problematic areas described here are by no means unique to Erlang and Java — other current languages such as Ada 95, Standard ML, Scheme, and C++, would perhaps serve the purpose of illustration even better. In fact, none of the programming problems mentioned above are especially new from a historical perspective.

But while issues like reactivity, concurrency, and communication traditionally have been the concerns of relatively remote fields like systems programming only, modern software demands have put such issues at the very core of almost any kind of software project. The once so distinct borders between programs, libraries, and operating systems are being blurred, which is currently causing radical changes in programming practice even for general purpose software. And as the complexity of general software increases, so does the need for programming methods that do not only facilitate, but moreover are able to *uphold* durable and flexible software structures. What the exposition above has tried to convey is that modern programming languages have not really kept pace with this trend.

1.2 Thesis

In this dissertation we put forward a programming language design where reactivity, concurrent objects, polymorphism, subtyping, and declarative properties are the prime concerns. Specifically, as the main thesis of our dissertation we claim that:

1. Active support for reactivity, declarative programming, concurrent objects, and safe polymorphic subtyping can be achieved in a single coherent programming language.

2. Such a language can be efficiently implemented.

3. The programming style of this language is comprehensible, often close to established practice, and well-adapted for the task of modern software construction.

To support our first claim we present a programming language, *O'Haskell*, with the desired characteristics. The semantic core of this language is formally defined, and its central properties are substantiated by detailed proofs.

The second and third claims are backed up by a concrete O'Haskell implementation, along with a number of programming examples written using this system. Although our interpreted implementation certainly is not optimized for efficiency, it clearly shows that implementing O'Haskell does not raise any

fundamentally new technical issues. Moreover, while the true usefulness of a language design can only be evaluated on basis of substantial real world testing, we will argue that our examples, of which several are non-trivial coding exercises, provide initial evidence that O'Haskell indeed possesses the merits that we claim.

O'Haskell is the result of a finely tuned combination of ideas and concepts from functional, object-oriented, and concurrent programming, that has its origin in the purely functional language Haskell [Pet97]. The name *O'Haskell* is a tribute to this parentage, where the *O* should be read as an indication of *Objects*, as well as a reactive breach with the tradition that views *Input* as an active operation on par with *Output*. The main design ideas underlying O'Haskell are:

- A minimalistic approach to object-based concurrency, which collapses the notions of an object and a process into a *reactive object*. Reactive objects are asynchronous, state-encapsulating servers whose purpose is to react to input messages; they cannot actively block execution, nor can they selectively filter their sources of input.

- A monadic integration of computational effects, which clearly separates stateful objects from stateless *values*. Apart from higher-order functions and recursive data structures, the latter notion also includes first-class commands, object templates, and methods.

- A safe, polymorphic type system with declared record and variant subtyping, supported by a powerful partial type inference algorithm.

A recurring theme in the definition of O'Haskell is a strive for simplicity. So is, for example, the concurrency semantics of O'Haskell much less complex than Java's threads, and its polymorphic subtyping system much closer to simple subtyping/polymorphic systems than what is the rule in most other approaches. We take this as an indication that O'Haskell represents a quite natural "fit" between the programming paradigms it attempts to combine.

In the remainder of this chapter we will develop and motivate the ideas behind O'Haskell in a little more depth (sections 1.3 to 1.5). We will also make comparisons with related language designs (section 1.6), and summarize our own contributions (section 1.7). The chapter ends with a synopsis of the material that constitutes the subsequent chapters of the thesis (section 1.8).

1.3 Reactive objects

1.3.1 Object modeling

The *object model* [RBP+90] offers a remarkably good strategy for decomposing a complex system state into a web of more manageable units: the state-encapsulating, identity-carrying entities called objects. Objects are abstractions of autonomous components in the real world, characterized by the shape of their

internal state and the *methods* that define how they react when exposed to external stimuli. The object model thus inherently recognizes a defining aspect of interactive computing: systems do not terminate, they just maintain their state awaiting further interactions [Weg96]. Not surprisingly, object modeling has become the strategy of choice for numerous programming tasks, not least those that must interface with the concrete state of the external world. For this reason, objects make a particularly good complement to the abstract, stateless ideal of functional programming.

The informal object model is naturally concurrent, due to the simple fact that real world objects communicate and "execute their methods" in parallel. On this informal plane, the intuition behind an object is also entirely reactive: its normal, passive state is only temporarily interrupted by active phases of method execution in response to external requests. Concurrent object-oriented languages, however, generally introduce a third form of state for an object that contradicts this intuition: the active, but *indefinitely indisposed* state that results when an object executes a call to a disabled method.

The view of indefinite blocking as a transparent operational property dates back to the era of batch-oriented computing, when interactivity was a term yet unheard of, and buffering operating systems had just become widely employed to relieve the programmer from the intricacies of synchronization with card-readers and line-printers. Procedure-oriented languages have followed this course ever since, by maintaining the abstraction that a program environment is essentially just a subroutine that can be expected to return a result whenever the program so demands. Selective method filtering is the object-oriented continuation of this tradition, now interpreted as "programmers are more interested in hiding the intricacies of method-call synchronization, than preserving the intuitive responsiveness of the object model".

Some tasks, like the standard bounded buffer, are arguably easier to implement using selective disabling and queuing of method invocations. But this help is deceptive. For many clients that are themselves servers, the risk of becoming blocked on a request may be just as bad as being forced into using *polling* for synchronization, especially in a distributed setting that must take partial failures into account. Moreover, what to the naive object implementor might look like a protocol for imposing an order on method invocations, is really a mechanism for *reordering* the invocation-sequences that have actually occurred. In other words, servers for complicated interaction protocols become disproportionately easy to write using selective filtering, at the price of making the clients extremely sensitive to temporal restrictions that may be hard to express, and virtually impossible to enforce.

Existing concurrent object-oriented languages tend to address these complications with an even higher dose of advanced language features, including path expressions [CH74], internal concurrency with threads and locks [GR85, AO93], delegated method calls [Nie87], future and express mode messages [YSTH87], secretary objects [YT87], explicit queue-management [Nie87, TA88], and reification/reflection [WY88]. O'Haskell, the language we put forward in this thesis, should be seen as a minimalistic reaction to this trend.

1.3.2 Preserving reactivity

The fundamental notion of O'Haskell is the *reactive object*, which unifies the object and process concepts into a single, autonomous identity-carrying entity. A reactive object is a passive, state-encapsulating server that also constitutes an implicit critical section, that is, at most one of its methods can be active at any time. The characteristic property of a reactive object is that its methods *cannot* be selectively disabled, nor can a method choose to wait for anything else than the termination of another method invocation. The combined result of these restrictions is that in the absence of deadlocks and infinite loops, objects are indeed just *transiently* active, and can be guaranteed to react to method invocations within any prescribed time quantum, just given a sufficiently fast processor. Liveness is thus an intrinsic property of a reactive object, and for that reason, we think O'Haskell represents a quite faithful implementation of the intuitive object model.

Concurrent execution is introduced in O'Haskell by invoking *asynchronous* methods, which let the caller continue immediately instead of waiting for a result. A blocking method call must generally be simulated by supplying a *callback* method to the receiver, which is the software equivalent of enclosing a stamped, self-addressed envelope within a postal request. Two additional features of O'Haskell contribute to making this convention practically feasible. Firstly, methods have the status of first-class values, which means that they can be passed as parameters, returned as results, and stored in data structures (see section 1.4). Secondly, the specification of callbacks, their arguments, and other prescribed details of an interaction interface, can be concisely expressed using the statically safe type system of the language (section 1.5).

There is, however, an important subcase of the general send/reply pattern that does not require the full flexibility of a callback. If the invoked method is able to produce a reply in direct response to its current invocation, then the invoking object may safely be put on hold for this reply, since there is nothing but a fatal error condition that may keep it from being delivered. For such cases, O'Haskell offers a *synchronous* form of method definitions, thus making it possible to syntactically identify the value-returning methods of foreign objects that *truly* can be called just as if they were subroutines.

With these contrasting forms of communication in mind, we may draw an analogy to the cost-effective, stress-reducing coordination behaviour personified by a top-rated butler(!). Good butlers ask their masters to standby only if a requested task can be carried out immediately and under full personal control (or, if necessary, using only the assistance of equally trusted servants). In all other cases, a good butler should respond with something like an unobtrusive "of course" to assure the master that the task will be carried out without the need for further supervision, and in applicable cases, dutifully note where and how the master wants any results to be delivered. Only an extraordinarily bad butler would detain his master with a "one moment, please", and then naively spend the rest of the day waiting for an event (e.g. a phone call) that may never come.

Reactive objects in O'Haskell allow only "good butlers" to be defined. We believe that this property is vital to the construction of software that is robust even in the presence of such complicating factors as inexperienced users, constantly changing system requirements, and/or unreliable, long-latency networks like the Internet. As an additional bonus, reactive objects are simple enough to allow both a straightforward formal definition and an efficient implementation. Still, the model of reactive objects is sufficiently abstract to make concurrency an intuitive and uncomplicated ingredient in mainstream programming, something which cannot be said about many languages which, like Java, offer concurrency only as an optional, low-level feature.

1.3.3 Reactivity in interfaces

One consequence of the restricted synchronization mechanisms of O'Haskell is that changing the implementation of a service, from a strictly local computation to a request involving network access, cannot be done without modifying the interface to that service. For example, if a service previously was implemented as as synchronous request, making the result dependent on packets arriving over a network can only be achieved by turning the service into an asynchronous method which takes a parameterized callback method as an argument. We argue that this is as it should be, since the semantics of the service also changes, from that of a simple subroutine call, to the uncertainty inherent in a synchronizing operation.

Such a change in semantics can have, and should have, consequences to the whole structure of the user of a service. This fact is well illustrated by programs that utilize Sun's distributed file system NFS. For pragmatic reasons, Sun decided not to change the file system interface when going from a local implementation to a distributed design, sensitive to partial failures. Since file system clients accordingly did not need to change either, most programs that access NFS remain totally unprepared for the fact that a remote file server may go down, which shows up to the user as a *total* loss of responsiveness in those situations [WWWK94]. Another example on the same topic is given by common web-browsers. These programs are generally carefully designed to allow continuous interaction even if a remote web server is not responding. However, most browser implementations fail to notice the fact that name-lookup of web addresses usually takes the shape of a remote service as well, and hence their user-interfaces freeze completely in the event of an inaccessible name server. Such a mistake would not be possible in a language where indefinite blocking cannot be masqueraded as a simple method call.

In fact, revealing the potential for indefinite blocking by requiring a callback is just a continuation of the argument that operational properties should be captured by types. Monadic programmers are already used to the idea that changing a pure function by adding side-effects should not be possible without simultaneously changing the type of the function as well. Reactive objects are simply an application of this idea to the *temporal* aspects of computing.

Having that said, we would also like to point out that even established

object-oriented practice is characterized by a desire to keep the message-passing metaphor free from blocking complications as far as possible, e.g. by attempting to concentrate all uses of potentially blocking operations to the head of a so-called *event-loop*. Programming styles developed with these considerations in mind are thus immediately expressible in O'Haskell, even without the introduction of callbacks. In fact, many common services do not even return a reply (e.g. methods that implement event-signaling or simple data pipelining), so in these cases simple asynchronous communication would suffice. The consequence of preserving reactivity primarily manifests itself in the encoding of classical synchronization primitives like semaphores and channels. Programming with explicit synchronization operations seems to require a major rethink in O'Haskell, a topic we will have reason to come back to in the subsequent chapters.

1.3.4 Preserving message ordering

A natural consequence of not allowing selective message reordering is to prescribe that methods should be executed in the order they are invoked, whenever that order can be uniquely determined. This is also exactly what O'Haskell does; i.e. an object can count on that two consecutive messages to the same peer will be handled in the order they are sent. However, common practice in concurrent languages is to leave this issue unspecified, in order to facilitate distributed implementations across an unreliable network [Agh86, PGF96]. We take on our opposing standpoint on basis of the following arguments.

1. Many simple programs would be unduly complicated if message ordering was not preserved (just imagine communicating with a storage-cell under such premises).

2. Processes on an unreliable network can be conveniently accessed via a library of local *proxy* objects. These proxies can present a reliable or an unreliable connection, at the free choice of the implementor. Implementing a proxy is greatly simplified, however, if communication with the local clients is order preserving.

3. The only operation which cannot be faithfully simulated by a proxy is the synchronous request. Thus, a non-local object will need to have a more limited interface than a corresponding local one. This is a necessary restriction if we consider liveness to be important, and indefinitely blocking input to be harmful. On the other hand, if the network in question is considered so reliable that the lack of synchronous requests is just perceived as an annoying constraint, then the network can equally well be made a part of the language implementation, since it guarantees the language semantics.

1.4 Values vs. Objects

1.4.1 Values in object-oriented languages

Most object-oriented languages provide very little support for data structuring tools besides objects. The reason for this sparseness is of course conceptual economy (the *everything-is-an-object* metaphor), but it also has as a consequence that every data entity must have the properties inherent in the object model, i.e. a *state*, an *age*, a *location*, an *identity*. For some kinds of data this point of view is entirely inappropriate — the value 7, for example, was not "created" at some point in time, neither can we make it cease to exist, nor count how many 7's there are. Moreover, adding 2 to 5 does not update the meaning of 5 or 2 in any way, nor does it produce a "new" 7. What mathematics teaches us is that numbers are abstract, timeless entities, and that computing with numbers (for example by adding them) is just a mechanical way of simplifying an expression to the canonical form that denotes its value.

Accordingly, all but the most spartan object-oriented languages give basic types such as numbers some special treatment that more closely matches our intuition. But there is a wealth of mathematically inspired structures that suffer from the same conceptual mismatch — these include pairs, lists, records, functions, and algebraic datatypes. Encoding such structures in terms of stateful objects is not only tiresome and error-prone, it may also prohibit efficient and safe sharing of data.

As an example, consider an encoding of vectors in space in terms of objects. In such an encoding, arbitrary vector values cannot simply be written as canonical expressions when needed, because values are represented by object references that must be created at some point in time by an instantiation command. Moreover, ordinary equality on these references actually makes it possible to discriminate values on basis of their location, i.e. to distinguish between *this* origin and *that* origin, etc. Hence the programmer must make sure that special purpose code gets called whenever a mathematical notion of vector equality is desired. And not the least, the programmer must also resist the temptation to implement a vector operation by destructively updating the state of the called object. If not, the meaning of vector values will indeed vary with the operations being performed, which is just as illogical for vectors as it is for numbers. Yet in an imperative language, destructive updating may often be the only reasonable implementation alternative, especially in the encoding of slightly more complex structures such as dynamic lists. Sharing stateful values with unknown code is obviously unsafe from this point of view, all the more if the unknown code happens to be a concurrently executing process. So this leaves the programmer with the choice of either conservatively cloning local objects that represent invariant values before sending them off to an unknown context, or (which is more likely) circumscribing stateful messages with informal restrictions on what can be done with them, and then simply hope for the best.

1.4.2 Functional programming

In contrast to the problem of finding good representations for values in object-oriented languages, functional programming builds directly upon a recognition of the mathematical nature of values. The functional paradigm advocates declarative construction, analysis, and simplification of arbitrary complex value expressions as the primary means of expressing a computational task. Variables in functional languages always denote constants, and in purely functional languages like Haskell, simplification of an expression is guaranteed not to have any computational effect whatsoever, besides termination with a canonical result. Among other things, this simple semantics opens up many possibilities for liberal sharing of data, since no program activity can affect the meaning of a variable, and no variable can be introduced that affects the meaning of a given program.

This property also goes under the name of *referential transparency*, and is often considered to be the defining characteristic of a purely functional language. Referential transparency enables the use of equational reasoning as a program verification technique, which indeed is a very strong argument in favour of a declarative programming style. Additional benefits commonly attributed to functional programming include a succinct notation, an efficient form of declaration by pattern-matching, and the important ability to treat functions themselves as first-class data values [Tur81].

Purely functional programs are however inherently weak in expressing interaction. Their calculator-like model of computation intuitively suggests that *input* is a single value that somehow happens to be available at program initiation, and that *output* can be limited to a single value returned at termination. Interacting objects can at best be modeled indirectly, but then only by means of encodings that reintroduce terms like state, change, and identity in the model — i.e. the very notions that advocates of functional programming deliberately have shun. Besides that, a fundamental problem with the functional model is that it cannot host the *non-determinism* of concurrently executing objects without a substantial repercussion on either its semantics or pragmatics [Mor98, Bur88].

The functional and object-oriented paradigms achieve conceptual efficiency by means of purification in terms of *either* values or objects. Yet both views are indispensable in modern programming [Mac82]. Values and value-oriented programming capture the fact that the computer indeed is a fast and flexible calculator, whereas objects and object-oriented programming focus on the storage capacity of computers, and the need for well-structured interaction with this storage. Value-oriented programming manifests computation by means of *expressions* that *denote a result*, while object-oriented programming uses *commands* that, when executed, will *cause an effect*. Values in this sense represent ideas, or abstractions in our minds, that have a name but no definition. Objects, on the other hand, represent our interpretation of concrete things in the world around us, as we understand them in terms of their constituent parts and their dynamic behaviour. The programming style for computing with values is thus inherently declarative, whereas the nature of object-oriented programming is

intrinsically imperative. And just as declarative *thinking* and imperative *doing* are complementary aspects of our daily life, so are the declarative and imperative styles when it comes to modern computer programming. Fortunately, the discovery of *monads* has made it possible for a programming language to be truly declarative and truly imperative at the same time. We will capitalize on this achievement in our integration of value- and object-oriented programming in O'Haskell.

1.4.3 A monadic approach to objects

Monads [Mog91, Wad92] are becoming widely recognized as the de facto standard for conservatively extending a declarative language with computational effects. The monadic approach is predominantly associated with Haskell and its imperative I/O system [LPJ95], but the technique is by no means limited to this language or to a particular evaluation strategy [Aug98, Fil94], not even to languages that exclusively belong to the functional category [Llo95].

Monads have been successfully utilized to incorporate a quite diverse set of classically imperative features in Haskell, including traditional I/O [LPJ95], graphical user interfaces [CMV97, FJ95], first-class pointers [LPJ95], concurrent processes and synchronization variables [PGF96], and exceptions [PRH+99]. The common denominator in all these proposals is the unifying property of a *stratified semantics*, that limits the effect of imperative commands to the top-level of a program, while leaving the purely declarative semantics of expression evaluation unaffected. This means that a monadic command under execution may very well request the evaluation of an expression, but an expression being evaluated can never trigger the execution of a command.

O'Haskell brings the world of object-based concurrency into the realm of monadic functional programming. Compared to other monad-based Haskell extensions, O'Haskell represents a higher level of abstraction, by its use of objects instead of pointers, and methods instead of peeks and pokes as the basic imperative building blocks. The high-level flavour of O'Haskell is also evident in its syntax for monadic programming, that improves on Haskell by also taking the need for unconvoluted use of assignable state variables into account.

The essence of the monadic approach is the insight that *doing* something and merely *thinking about doing* something are radically different activities. Translated into programming terms, this means that an imperative command is also considered to be a first-class declarative value, although a value of an abstract type (the monad) that just represents the hypothetical effect of a command, *should it ever be executed*. Command values can thus be declaratively combined, named, parameterized, and calculated with just like any other value, since the evaluation of a command expression is kept entirely distinct from the actual *realization* of the effect it represents.

All commands in O'Haskell, including those that refer or assign to state variables, enjoy the status of first-class values. Furthermore, since any command can be individually named, the name of a method invocation may equally well be considered the name of the actual method it invokes, just as the name of an

object creation command may be identified with the *class* of objects it instantiates. Hence we are able to promote even methods, as well as object templates,[1] to full-blown value status in O'Haskell.

This last property makes parameterization over callback methods easy. In fact, higher-order parameterization as a general programming technique has the potential of facilitating very simple solutions to many problems that immediately seem to lead to quite contorted encodings in established object-oriented languages. Examples of this include parameterization of an object template over (i) a set of neighbouring objects, (ii) other templates from which private but unknown sub-objects may be created, or (iii) a single command denoting a yet undefined aspect of an object's behaviour. Other speculative examples are methods creating object instances from a supplied template parameter on behalf of external clients, methods taking a communication pattern as a command argument, or simply a method parameterized over a function value that expresses some unknown computation on the local state. This flexibility should be compared to the complicated exercise in inheritance and override that is required in an object-oriented language like Java to express even such a simple concept as a callback parameter.

The communication interface returned when an object is created is also a full-fledged value in O'Haskell, that may or may not reveal information about the identity of the object behind the interface. Most likely an object interface is defined as a record of method values, but it might also be a tuple of such records, or even a function that returns different interfaces depending on a supplied argument such as a password! Separating values from stateful objects opens up many interesting possibilities, as we hope our coding examples will demonstrate.

1.5 Subtyping and polymorphism

Type systems for functional and object-oriented languages have evolved along two quite orthogonal lines. Mainly due to the influential type system of Standard ML [MTH90], functional type systems in general have acquired a certain flavour that can be characterized by a preference for the algebraic datatype (the *labeled sum*) as the principal data structure, and a widely employed use of *parametric polymorphism* as the primary means of achieving typing flexibility. This is matched in modern object-oriented type systems by an equally strong preference for data structures based on records (*labeled products*), and a tradition of obtaining flexible typings through the use of *subtyping* (again mostly due to a single seminal language: Simula [DMN68]). Other, mostly one-dimensional distinctions can also be identified, for example concerning the questions of automatic type inference vs. explicit typing, or static type safety vs. optional run-time type inspection. Still, these minor observations do not

[1] O'Haskell actually uses the term *template* instead of class to avoid confusion with the established Haskell concept of *type classes*.

really contradict the fact that functional and object-oriented type system are essentially characterized by features that are by no means mutually exclusive.

The predominance of a particular kind of type-forming operator in each school is not a coincidence, though [Coo91]. Programming is essentially the task of manipulating data structures, and different parts of a program take the roles of either data-producers or data-consumers in this play. The functional approach to programming is to ask "how is data constructed?". This leads to a style of programming where the data *constructors* are considered primitive, and where data-consuming code accordingly is defined by pattern-matching over these constructors. The object-oriented programmer instead starts out by asking "what can we do with the data?", which fosters the view that it is the data *selectors* that should be seen as foundational, and that the code of a data-producer consequently should be defined as an enumeration of implementations for these selectors.[2]

The preference for a certain form of data structures also reflects a particular view of where extensibility is most needed. The functional style ensures that adding another data-consumer (a function) is a strictly local change, whereas adding a new data-producer requires extending a datatype with a new constructor, and thus a major overhaul of every pattern-matching function definition. Likewise, in object-oriented programming the addition of a new data-producer (a new class) is a cinch, while the introduction of another data-consumer means the extension of an record type with a new selector, a global undertaking with implications to potentially all existing classes.

To alleviate these drawbacks to some extent, functional and object-oriented traditions prescribe their own custom methods. Parametric polymorphism in functional languages allows functions to be defined uniformly for arbitrary sets of data constructors, including constructors not yet visualized. Object-oriented subtyping, on the other hand, facilitates incremental definition of record types, so that old data-consuming code still can be reused even on data objects defined by a larger set of selectors than what was originally envisaged. The latter openness to future refinements in design is actually the intuition behind the unmistakable *is-a*-relation that forms a central part of the biological metaphor permeating object-oriented classification [Weg87]. However, as is well known, nothing really precludes the merits of subtyping from being applied to the definition of datatypes as well; or for that matter, polymorphism to be utilized in conjunction with records [CW85].

As a hybrid language, O'Haskell attempts to bring the design issues that accompany constructor- or selector-based programming to the fore, so that the programmer can choose the right kind of data structure for each task. Hence the type system of O'Haskell offers equal opportunities for working with labeled sums and labeled products. Polymorphism is available in both cases, and so is the possibility of defining types incrementally, (i.e. subtyping is supported

[2]Here already lies the key to the distinctive form of data abstraction that goes under the name object-orientation: the producer of a data object gets full control over what the result of consuming the object will be — the consumer is effectively hindered from making that decision on basis of how the object was constructed.

for both parameterized records and parameterized datatypes). Furthermore, neither kind of data structure is tied to any particular choice between declarative and imperative programming, nor is the availability of type inference dependent on whether a datatype or a record type is used. However, type inference in conjunction with subtyping holds its special set of problems, which have been given a somewhat unconventional, yet very effective solution in O'Haskell.

1.5.1 Subtyping and type inference

The combination of subtyping with polymorphic type inference has been under intensive study for more than a decade [Mit84, FM90, FM89, Mit91, Kae92, Smi94, AW93, EST95, Hen96, Reh97, MW97, Seq98]. From the outset, this line of research has focused on *complete* inference algorithms, and the related notion of principal types. This direction is not hard to justify considering the evident merits of the Hindley/Milner type system that underlies polymorphic languages like Haskell or Standard ML: program fragments can be typed independently of their context, and programmers may rest assured that any absent type information will be filled in with types that are at least as general, and at least as succinct, as any information the programmer would have come up with.

Still, although complete algorithms for polymorphic subtype inference exist, practical language implementations that take advantage of these are in short supply. The main reason seems to be that "the algorithms are inefficient and the output, even for relatively simple input expressions, appears excessively long and cumbersome to read" [HM95]. Many attempts have been made at simplifying the output from these algorithms, but they have only partially succeeded, since the problem in its generality seems to be intractable, both in theory and practice [HM95, Reh97].

However, even if type simplification were not an issue, there is an inherent conflict between generality and succinctness in polymorphic subtyping that is not present in the original Hindley/Milner type system. While the principal type of (say) a Standard ML expression is also the syntactically shortest type, the existence of subtype constraints in polymorphic subtyping generally makes a principal type *longer* than its instances. In particular, the principal type for a given expression may be substantially more complex than the simplest type general enough to cover an *intended* set of instances! Thus, type annotations, which give the programmer direct control over the types of expressions, are likely to play a more active role in languages with polymorphic subtyping than they do in Haskell or Standard ML, irrespective of advances in simplification technology.

In the design of a type system for O'Haskell, we have taken a pragmatic standpoint and embraced type annotations as a fact of life. This has enabled us to focus on the much simpler problem of *partial* polymorphic subtype inference. As one of the main contributions of the thesis, we present an inference algorithm for our type system that always infers types without subtype constraints, if it succeeds. This is a particularly interesting compromise between implicit and explicit typing, since such types possess the desirable property of being

syntactically shorter than their instances, even though they might not be most general. We might say that the algorithm favours readability over generality, leaving it to the programmer to put in type annotations where this strategy is not appropriate.

Our inference algorithm is based on an approximating constraint solver, that resorts to ordinary unification when two type variables are compared. An exact characterization of which programs the algorithm actually is able to accept is still an open problem, but we prove, as a lower bound, that it is complete with respect to the Hindley/Milner type system, as well as the basic type system of Java. The algorithm is furthermore both efficient and easy to implement, and as our programming examples will indicate, explicit type-annotations are rarely needed in practice.

1.5.2 Name inequivalence

An additional contribution of the O'Haskell type system is presumably the subtype relation itself, which is based on *name inequivalence*, in contrast to the structural ditto that dominate the standard literature [CW85, Mit84]. By structural we mean the common practice of defining special subtyping rules for functions, records, and variants, etc, assuming a given partial order over a set of nullary base types. Being a conservative extension to the type system of Haskell, our system instead assumes that type constants can be of any arity, and the partial order on base types is accordingly replaced by a relation on fully saturated applications of these constants. Our subtyping relation furthermore supports polymorphic instantiation of the basic subtype axioms, and a notion of *depth* subtyping that works inside type constant applications on the basis of inferred *variances* for the type parameters.

We believe that working with named and parameterized types, whose subtype relationships are defined by declaration, has the immediate benefit of giving the programmer full control over the type structure of a program. This can be a valuable tool in the design phase of a large system, and it also offers greater protection from certain logical errors. Furthermore, name inequivalence is more in line with both the way datatypes are treated in Haskell and other functional languages, and with the object-oriented notion of subtyping between named classes. And not the least, the ability to refer to types by name has a big impact on the readability of the output from our inference algorithm.

1.6 Related work

1.6.1 Concurrency in functional languages

Much work has been done in extending functional languages with concurrency features. Within the lazy functional community, *stream-based* approaches have a fairly long tradition [Hen82, Sto86, Tur87, HC95]. A common characteristic of these solutions is that communication is directed towards a particular *process*,

and that all input streams for a process are merged into one before reception. This means that stream-based processes do not constrain the order in which different messages are received, a feature quite similar to the reactive property of our objects. The drawback of the stream-based school is that it requires a rather heavy use of disjoint sum types in order to make complex message routings work, and that the coupling between output and input is very loose, thus ruling out synchronous constructions like our requests.

A different approach is to separate the concept of a process and a communication destination. This is the common view in most process calculi, and it has been adapted in many functional languages as well, including *Concurrent Haskell* [PGF96], *Concurrent ML* [Rep92], *PFL* [Hol83], *Facile* [GMP89], and *Pict* [PT98b]. An immediate benefit of these systems is that most message typing problems disappear, even for very complex communication patterns. However, since a *channel* (as we might collectively call the passive communication points) must be addressed explicitly in both the send and receive operations, the ability to simultaneously wait for any kind of message is lost. One remedy is to introduce the *choice* operator known from process calculi [Mil91], but its complexity generally makes it hard to implement efficiently, and great care must also be taken to avoid loss of abstraction when it is used [Rep92]. Choice-free encodings of object-like structures using multiple threads and locks are described in [PT94], while [PGF96] gives an example of the standard way of tagging messages by means of a datatype in order to encode iterated choice. O'Haskell contrasts to both these approaches by offering direct support for synchronous and asynchronous messages of multiple types, without the need for multiplexing by tagging, or coordination by additional processes.

Erlang and *UFO* are two concurrent functional languages with a communication model similar to that of O'Haskell, where the addressee of a message is a process instead of an anonymous channel [AVWW96, Sar93]. Erlang also endorses a sequential view of processes like we do, but relies on tail calls to keep a process alive, and utilizes an untyped, Prolog-like pattern-matching mechanism to specify a current set of accepted messages. UFO is typed and has encapsulated state variables, but the object/process correspondence is vague and sequencing comes from data dependence only. Message acceptance can furthermore be restricted by manipulating predefined pseudo-variables in UFO. Neither of these languages can be called reactive in our meaning of the word, although common practice in Erlang programming seems to be to manually assert that every process always can handle any message [AVWW96].

The purely functional language *Concurrent Clean* comes equipped with an event-driven I/O library modeled after the influential Macintosh Toolbox [Ach96]. However, this library does not specifically address any reactivity issues, and the support for concurrent objects is very weak. Clean is further distinguished by its rather cumbersome way of coding imperative programs by threading an explicit "world state" parameter through every stateful computation.

Despite its different communication model, Concurrent Haskell holds a unique position in relation to O'Haskell since it is the only concurrency extension

to Haskell in widespread use, and because its monadic approach to concurrency has had a strong influence on our work [PGF96]. These influences are clearly visible in our development of a semantics for reactive objects, especially in the choice of notation. The main semantic difference is that the meaning of `return ()` is a terminated process in Concurrent Haskell, whereas the same expression stands for an *inactive* process, subject to future activation, in our formalism. A detailed comparison in terms of bi-directional translations between the two languages can be found in chapter 6. See also the issues on reactivity raised in subsection 1.6.3.

An interesting characterization of concurrency proposals for functional languages is of course whether they are able to preserve a pure semantics of expression evaluation. Of the works cited above, [HC95, Tur87, Sto86, PGF96, Hol83, Ach96] belong to this category, while [Hen82, Rep92, AVWW96, Sar93, PT98b] do not.

1.6.2 Concurrent object-oriented languages

Our unified view of objects as processes stems from the *Actor* model [Agh86, AMST97], which can be characterized as the original source of many, perhaps even most, concurrent object-oriented languages. Like objects in this model, an O'Haskell object can basically do three things in reaction to a message: (1) send messages, (2) create objects, and (3) update its local state. In the Actor model reported in [Agh86], these activities all occur in parallel, and reaction is thus atomic. Whether reaction can be seen as atomic in O'Haskell depends on the time-scale, but the computation model we employ for method bodies is in any case strictly *sequential*, as a consequence of the monadic approach.

An alternative Actor semantics that embodies a sequential functional language is described in [AMST97]. This system is actually quite close in spirit to O'Haskell, despite the apparently non-monadic style. The given reduction system for Actor configurations also reflects this similarity: like in the formalization of O'Haskell, objects/processes are terms tagged with unique addresses, and the reduction rules for the primitive constants match against contexts that single out the head of an otherwise unreducible sequence of applications. However, differences are also clearly identifiable. While state change in the Actor model is equivalent to specifying a replacement behaviour, O'Haskell relies exclusively on state variables. This has the desired effect of making the state of an O'Haskell object clearly decoupled from the set of messages it is set up to handle. O'Haskell also makes stronger assumptions about the order of message deliveries than does the Actor model; another distinction is that the latter formalism does not provide anything similar to our synchronous requests.

ABCL/1 [YSTH87] and *Concurrent Smalltalk* [YT87] are two concurrent object-oriented languages that also exhibit similarities to O'Haskell: processes are identified with an encapsulated state, objects rest between (mutually exclusive) method invocations, and both synchronous and asynchronous forms of communication are supported. However, O'Haskell notably lacks the complex synchronizing and parallelism-enhancing features that the other languages seem

to rely on: explicit message reception in ABCL/1 and secretary objects in Concurrent Smalltalk for selective method disabling, future messages/CBox objects in the respective languages for circumventing blocking method invocations, and express mode messages in ABCL/1 for interrupting objects stuck in a loop or a method call. On the other hand, O'Haskell has the advantage of being higher-order and strongly typed, which guarantees a much smoother adaption of alternative programming techniques, such as parameterization over callbacks, than what can ever be the case in the mentioned first-order and untyped languages.

An approach to object-oriented concurrency that matches O'Haskell in simplicity can be found in *Emerald* [RTL+91]. Like O'Haskell this language associates a thread with each object, makes a clear separation between interface types and implementation definitions, lacks any complex notion of inheritance, and is strongly typed. The similarities end here, though, since the thread of an Emerald object is not given exclusive access to its state. Instead all methods execute within the thread of the caller, and encapsulation as well as synchronization must be explicitly handled by means of an optional *monitor* construct, that also needs to involve the private thread of an object. Furthermore, even though the type system of Emerald includes notions of both subtyping and polymorphism, the approach taken is quite non-standard, due to an underlying "types-are-objects" principle.

Objects in O'Haskell definitely share some characteristics with the monitor concept as such, though. Monitors were introduced by Hoare to automatize the typical pattern where a *semaphore* is always released by the same process that has claimed it, as a means of protecting a critical section [Hoa74]. The other distinct use of a semaphore, as identified by Dijkstra in [Dij68], feeds input to a particular process by means of a counting semaphore which is exclusively claimed by the receiver. This latter requirement is handled in a monitor by explicitly managed *condition variables*, something which significantly complicates the monitor semantics. We may say that O'Haskell generalizes the monitor concept by allowing some protected operations to be invoked asynchronously, at the same time as it simplifies the same concept by replacing condition variables with support for first-class use of the monitor operations themselves.

1.6.3 Reactivity

Current work on reactive languages is mostly concerned with the *synchronous* approach to reactivity [BB91, BG92, BDS96, BS98]. In this school, all activations in a system are synchronized by a global logical clock in order to obtain deterministic concurrency, which is crucial in hard real-time systems. The main hypotheses made in the synchronous model are that computations take zero time and event delivery is instantaneous. From these assumptions it follows that events received or generated somewhere between two clock ticks are actually occurring *simultaneously*, and hence, for example, multiple invocations of a particular method during an instant must be indistinguishable from just a single invocation [BDS96].

In O'Haskell there is no formal notion of a global instant, as every object

executes at its own speed. Still, for a specific object it is definitely meaningful to speak in terms of instants, since execution cannot block indefinitely between two passive states. So under the assumptions that processing is sufficiently fast and that objects do not loop or send each other infinitely many messages, an O'Haskell system as a whole can at least informally be understood as alternating between stable, resting states and outbursts of instantaneous execution. What O'Haskell lacks compared to the synchronous formalisms is the ability to collapse "simultaneous" method calls into one — no matter how close in time, messages still arrive in some specific order in O'Haskell, and if a receiving object cannot be considered to react to these instantaneously in reality, execution will simply lag behind without any effect on the other objects in the system.

In part we have chosen this asynchronous approach because it adheres more naturally to the intuition of an established message-passing metaphor, and hence involves a less esoteric programming style. Our main motive, however, is that we have a broader class of applications in mind, including graphical user-interfaces, databases, network protocols, and operating systems, for which it would seem unfortunate to tie the timing of all activations to the slowest component of the system. This observation has also been made by other developers working with asynchronous languages; see for example the work on the *Triveni* toolkit for Java [CJJ+98a, CJJ+98b].[3]

In the domain of functional languages, reactive libraries have been proposed by Pucella [Puc98] and Scholz [Sch98]. They both adopt the synchronous view in order to obtain deterministic parallelism, although Pucella does not really provide the communication mechanisms where simultaneity really would make a difference. Scholz's proposal is still of some relevance to our work since it is written in Haskell and is entirely based on a monad, hence the purely functional semantics of the host language is left unaffected. A related formalism, which models time as continuous instead of divided into instants, is the reactive animation toolkit *Fran* [Ell97]. However, both behaviours and events are basically just functions over time in Fran, and its support for reactive programming appears to have limited value outside the specific area of animation.

In [FJ95] and [FJ94] Finne and Peyton Jones argue forcibly *against* the event-driven view of reactive systems, in favour of the more conventional model of Concurrent Haskell, where it is "the application that drives the I/O". The complications that accompany this view when multiple input sources are present are proposed to be addressed by an increased amount of concurrency and shared memory communication, so that each process can retain the view that input just comes from one single quarter.

While Finne and Peyton Jones correctly identify the problems with existing frameworks based on *event-loops* (e.g. the centralized program structure and the bulkiness of explicit interleaving), they come to a precipitate conclusion what regards event-driven programming, since what they criticize is not primarily the event-driven style, but more the lack of concurrency and state en-

[3] We also note that asynchronous languages like Erlang have been successfully applied in the construction of *soft* real-time systems [AVWW96].

capsulation in common frameworks. Indeed, the solutions the authors provide, namely monadic concurrency and asynchronous scheduling of events, are actually cornerstones in the event-based approach of O'Haskell as well. Moreover, the demanded partitioning of a global system state into more decentralized units is almost mandatory in our language. We think that O'Haskell shows that the choice for or against event-driven reactive programming must be made on other grounds than those expressed in the cited works; perhaps tentatively on basis of how the relative merits of the implied message-passing and shared memory communication models are judged.

In the field of computational models, the nondeterministic *input/output automaton* (IOA) model of Lynch and Tuttle takes a clearly event-driven reactive route with its *input enabled* processes [LT89]. The main difference relative O'Haskell is that events in the IOA model are defined as predicates on the local state of an automaton, and not something that is directly caused by an automaton executing an action. A closer comparison between O'Haskell and the IOA model will appear as part of a programming case study in chapter 4.

1.6.4 Polymorphic subtyping

Algorithms for polymorphic subtype inference are described in [Mit84, FM90, FM89, Mit91, Kae92, Smi94, AW93, EST95, FA96, Hen96, Pot96, TS96a, Reh97, MW97, Seq98, Pot98]. Strategies for simplification of the inferred constraint sets are discussed in [FM89, AW93, EST95, Reh97, MW97, Pot98] in particular. The choice to settle for a deliberately incomplete inference algorithm sets our work apart from most of the cited systems, though.

Smith and Trifonov [TS96a], and Pottier [Pot96, Pot98] approximate entailment checking with decidable variants that result in loss of information, but this move is not motivated by any desire to actually eliminate constrained types; the algorithms are rather designed to have a negligible effect in this respect. Aiken and Wimmers' constraint solver also makes some non-conservative, but hardly radical, simplifications in the interest of efficiency [AW93]. Reppy and Riecke, though, have implemented type inference for their object-oriented ML variant *OML*, in such a way that only constraint-free type schemes are inferred [RR96]. However, they do not describe their algorithm, nor do they provide any technical results.

Most related to our approach is probably Pierce and Turner's work on local type inference for Pict [PT98a]. They start with a variant of (the explicitly typed) *System F* with subtyping, and develop an inference technique that is able to fill in missing type arguments, as well as missing annotations, in many cases. Their method for inferring types for anonymous abstractions is similar to our implementation of integrated type-checking (see section 5.5), although their algorithm switches between strict phases of inference and checking, which ours does not. Pierce and Turner's algorithm is not complete w.r.t. the Hindley/Milner system, but they have the advantage of a declarative specification of all programs that the inference algorithm accepts.

Cardelli's implementation of $F_{\leq}$ [Car93] also contains a partial type inference algorithm that, like ours, uses unification to solve constraints involving unbound type variables. However, this "greedy" algorithm solves *all* constraints, including those on variables in the assumption environment, at the earliest stage possible, thus its behaviour can sometimes appear quite unpredictable.

1.6.5 Name inequivalence

Most of the work on subtype inference cited above take a *structural* view of the subtype relation, that is, types are subtype-related if and only if they have the same structure (a function, a pair, a list, etc) and their respective constituent parts are subtype-related. This leads to a subtyping theory where the only primitive subtype relationships are between nullary base types. The systems described in [Smi94, EST95, Seq98] follow [CW85] and allow two records to be subtype related if their set of selectors are in a *superset* relationship. Still, it is the structure of a record that defines its position in the subtype hierarchy, not any intentions explicitly declared by the programmer. Henglein's type system [Hen96] is unique in that it is *generic* in the choice of subtyping theory, something which we have exploited in our system based on *name* inequivalence.

Despite the fact that type systems with name-based (in)equivalence dominate among both functional and object-oriented languages currently in widespread use, polymorphic subtype inference for such systems have not gained much attention. Reppy and Riecke's OML employs a scheme where object types are related by declaration [RR96], while Jategonkar and Mitchell outline a similar (future) system for abstract datatypes in [JM93]. Both these systems consider only parameterless type constants. Sequeira allows user-defined type constructors with declared non-atomic upper bounds, but the type constructors themselves must always have zero arity [Seq98].

The language closest to O'Haskell in this respect is the Java extension *Pizza* [OW97], which allows classes to be parameterized on types, and which supports extension of parameterized classes in a way that resembles our polymorphic subtype axioms. There is also a notion of algebraic datatypes in Pizza, but here type extension is not supported. In contrast to O'Haskell, Pizza only supports *invariant* subtyping (i.e. equality) inside a parameterized type constructor, and the *subsumption* principle is not upheld; that is, if an expression of type A is expected and B is a subtype of A, then an expression of type B *cannot* be used in Pizza, except in the special case of method lookup. Pizza is moreover an explicitly typed language on the whole, although polymorphic identifiers are instantiated correctly without manual intervention. The completeness result for our inference algorithm w.r.t. the Java type system could probably be generalized to hold relative to a suitably adapted version of Pizza's type system as well, but proving this remains as future work.

Subtyping based on variance annotations can be found in the various object calculi by Abadi and Cardelli [AC96], although here the annotations are attached to method names instead of type constructor arguments. Freeman's work on refinement types involves inferred variances for datatype constructor

arguments [Fre94], but his setting is rather different from ours, in the sense that his subtyping system serves to provide additional information on terms that already have an ML type. The system $ML_{\leq}$ allows both variances and subtype orderings to be declared, but imposes the restriction that the variance as well as the arity must be uniform for all related type constructors [BM97]. We are not aware of any system besides ours that provides a combination of subtype declarations as general as our polymorphic subtype axioms, and variance-based depth subtyping.

Objective Caml (Objective ML) conservatively extends ML with subtype inference and a powerful type abbreviation mechanism that achieves much of the succinctness we obtain by using name inequivalence [RV97]. However, the type system of Objective Caml is based on extensible records (in contrast to extensible record *types*), and does not support subsumption.

Many of the systems mentioned allow subtype constraints to be recursive [Kae92, AW93, EST95, RV97, MW97, OW97], thus facilitating certain object-oriented programming idioms not otherwise expressible [BCC+96]. Our system does not currently support recursively constrained types, although a generalization that would allow such constraints to be *checked* (just as ordinary constraints can be checked but not automatically inferred by our system) would be an interesting future extension.

1.7 Contributions

The main contribution of this dissertation is the programming language O'Haskell, which embodies all the characteristics of a language suitable for modern software construction postulated in the beginning of this chapter: active support for reactivity, a declarative programming style, straightforward concurrent objects, and safe polymorphic subtyping. This result can further be broken up into the following specific achievements:

- A coherent language design, integrating central concepts from functional, object-oriented and concurrent programming in one language.
- A provably sound extension of a purely functional language with assignments, state-encapsulation, and non-deterministic, actor-like concurrency.
- Simultaneously, a sound extension of a simple concurrent object-oriented language with stateless data structures and first-class functions, commands, templates, and methods.
- An exploration of asynchronous reactivity as a major structuring tool in the design of interactive applications, embedded controllers, and network protocols.
- A provably sound polymorphic subtyping system with records, algebraic datatypes, name-inequivalence, and subtyping by declaration.

- A practically useful approach to polymorphic subtype inference by means of a partial inference algorithm, that is proven complete w.r.t. the basic type systems of Haskell and Java.
- A full language implementation in terms of the O'Hugs interactive system.

1.8 Outline of the thesis

The thesis is basically organized into two halves, where the first half presents O'Haskell from the programmer's point of view in terms of language constructs and programming examples, and the second half deals with the theoretical and technical sides of the language. The dividing line goes between chapters 4 and 5; ahead of that line Greek symbols will start to appear much more frequently. We will summarize each of the chapters in turn.

Chapter 2: Survey of O'Haskell Here we introduce O'Haskell in the tutorial form, by means of short programming fragments that illustrate the important features of the language. Some previous knowledge of Haskell or functional programming in general will of course be valuable here as in the rest of the thesis, but the chapter still starts out with a basic overview of Haskell which should be fairly complete as far as the subsequent material is concerned. Then the record and subtyping extensions are presented, followed by sections on automatic type inference and the support for reactive objects. Some broad remarks regarding the lack of a classical inheritance mechanism in O'Haskell are provided at the end.

Chapter 3: Programming examples In this chapter a series of more self-contained programming examples is presented. These studies include a typical interactive graphical application in the form of a drawing program, an embedded controller for an autonomous guided vehicle, a parallel algorithm for computing prime numbers, a network protocol layer in the form of a Telnet client, and an implementation of the central parts of a telephone exchange. All these examples demonstrate the natural connection between reactivity, concurrent objects, and environment interaction, with the exception of the prime number algorithm, which primarily serves to illustrate the support for true parallel programming in O'Haskell. The subtyping feature is especially manifest in the drawing example, which also contains a description of a strongly typed interface to the popular graphical toolkit Tk [Ous94]. The chapter moreover contains reactive encodings of some classic primitives that involve indefinite blocking, and we show how synchronization using such primitives can be conveniently expressed in O'Haskell by the supply of callbacks.

Chapter 4: Case study A more substantial programming case study is described in the chapter: a modular implementation of an algorithm for self-stabilizing network protocols by means of local checking and correction

[APSV91]. Apart from providing some insights into the original self-stabilizing algorithm, as well as the relation between O'Haskell and Input/Output Automata [LT89], the chapter primarily contributes an approach to programming network protocols in O'Haskell, that has a much wider applicability than to the field of self-stabilization alone. We also continue the discussion on inheritance in O'Haskell here, in the light of the undertaken case study.

Chapter 5: Subtyping Here we begin the formal development of O'Haskell, by studying the type system and its accompanying partial subtype inference algorithm. The static type system is defined as an extended instance of the type system described in [Hen96], and we show how the referenced work can be built upon to yield some characteristic results for our system, including a subject reduction property, and the existence of principal types. A precise definition of a subtyping logic based on name inequivalence is also given, and we show that typing judgements are invariant under subtyping in this logic. We furthermore give a precise description of the inference algorithm, and we prove its central properties: soundness, decidability, and completeness w.r.t. the core type systems of Haskell and Java. The chapter finally contains a formal account of top-down type checking in conjunction with our partial inference algorithm.

Chapter 6: Reactive objects In this chapter we provide a formal definition of the dynamic semantics of reactive objects. This undertaking is split into two steps: first a set of syntactic transformations narrow down the support for reactive objects to the existence of a set of primitive monadic constants, then these constants are given an operational semantics in terms of a non-deterministic reaction relation between object/process configurations. Two technical results concerning this reaction relation are given; one which shows that reaction steps preserve typing judgements, and one which characterizes our informal claim that the purely functional semantics of Haskell is not compromised by the inclusion of non-determinism and destructive assignments in our extension. A closer comparison with Concurrent Haskell [PGF96] is finally provided, which shows that the two languages can indeed implement each other's concurrency primitives.

Chapter 7: Implementation The technicalities of implementing O'Haskell are discussed in this chapter, in particular we describe the steps we have taken to turn the interactive Haskell system *Hugs* [Jon96a] into an implementation of O'Haskell. Issues regarding code generation and run-time data representations are also discussed, as well as potential improvements on these matters in a future optimizing compiler. Some notes are furthermore added on how the subtyping algorithm of chapter 5 is adapted to cover the full language as it is seen by the programmer.

Chapter 8: Conclusion and future work In this final chapter of the thesis we restate our results and draw some conclusions. On the subject of future work on O'Haskell we list both practical issues such as development of a more mature

implementation and collection of more programming field experience, as well as theoretical developments such as finding better characterizations of the type inference algorithm. Some possible improvements on the actual language are finally outlined, which include an idea on how to achieve better integration of the powerful overloading system inherited from Haskell.

Chapter 2

Survey of O'Haskell

In this chapter we provide a detailed, but informal survey of O'Haskell and its characteristic features. A formal semantic treatment of the language will appear in chapters 5 and 6; in this chapter the exposition will instead be based on short code examples in order to prepare the reader for the more substantial examples of O'Haskell programming that will follow.

Since O'Haskell is defined as a conservative extension to the purely functional language Haskell, it is natural to base the survey on some previous knowledge of this language and focus the discussion on the aspects where O'Haskell differs from its predecessor. However, in order to emphasize that O'Haskell is not intended as merely an internal affair within the functional programming community, we will start this chapter by summarizing O'Haskell from three different perspectives, each one representing one of the programming paradigms that the language attempts to combine:

- O'Haskell is an **object-oriented imperative language**, offering state encapsulation, identity-carrying objects, extensible interface hierarchies with subtyping, and the usual array of imperative commands like loops and assignments. Implementation inheritance in the style of e.g. SmallTalk is not directly supported, but this is largely compensated for by rich facilities for parameterization over first-class functions, methods, and templates. Additional O'Haskell features not commonly found in object-oriented languages include parametric polymorphism, automatic type inference, a straightforward concurrency semantics, and a powerful expression sublanguage that permits unrestricted equational reasoning.

- O'Haskell can furthermore be characterized as a strongly typed **concurrent language**, centered around a monitor-like construct with implicit mutual exclusion, and a message-passing communication metaphor offering both synchronous and asynchronous communication. Unlike most concurrency models, however, an O'Haskell process is identified with the notion of an encapsulated state (i.e. an object), and the execution thread of such a process is moreover not conceived as being continuous, but rather

broken up into non-blocking fragments whose mutual execution order is determined by its external, message-passing clients.

- O'Haskell is finally a **purely functional language**, with lazy evaluation, higher-order functions, algebraic datatypes, pattern-matching, and Hindley/Milner-style polymorphism. O'Haskell also supports type constructor classes and overloading as in Haskell, but these features play no central role in the definition of its extensions. To this base O'Haskell conservatively adds two major features: *subtyping*, and a monadic implementation of *reactive objects*. The subtyping extension is defined for records as well as datatypes, and is supported by a powerful partial type inference algorithm that preserves the types of all programs typeable in Haskell. The monadic extension is intended as a replacement for Haskell's standard IO model, and provides concurrent objects and assignable state variables without breaking referential transparency.

The rest of this chapter is organized as follows. Section 2.1 continues with a brief overview of the base language Haskell and its syntax, before we introduce the major type system additions of O'Haskell: records and subtyping (sections 2.2 and 2.3). In section 2.4 our approach to automatic type inference in O'Haskell is presented. Reactive objects, concurrency, and encapsulated state are the issues discussed in section 2.5. Section 2.6 presents some additional syntactic enhancements that O'Haskell provides, before the chapter ends with a section discussing the role of *inheritance* in O'Haskell programming (section 2.7).

2.1 Haskell

Haskell [Pet97] is the quintessential non-strict, purely functional language, and the base upon which O'Haskell is built. While we suspect that some previous acquaintance with Haskell might be necessary in order to get the most out of the present thesis, we will nevertheless take some steps to familiarize the reader with the syntax of Haskell in this section.

Functions

Functions are the central concept in Haskell. Applying a function to its arguments is written as a simple juxtaposition; that is, if `f` is a function taking three integer arguments, then

```
f 7 13 0
```

is an expression denoting the result of evaluating `f` applied to the arguments `7`, `13`, and `0`. If an argument itself is a non-atomic expression, parentheses must be used as delimiters, as in

```
f 7 (g 55) 0
```

Operators like + (addition) and == (test for equality) are also functions, but written between their first two arguments. An ordinary function application always binds more tightly than an operator, thus

```
a b+c d
```

should actually be read as

```
(a b) + (c d)
```

Laziness

Non-strict semantics means that function arguments are only evaluated when absolutely needed by a function (hence the epithet *lazy* is often used to describe non-strict languages). So, even if

```
g 55
```

is a non-terminating or erroneous computation (including for example an attempt to divide by zero), the computation

```
f 7 (g 55) 0
```

will nevertheless succeed in Haskell, provided that f happens to be a function which ignores its second argument (whenever the first one is 7, say). This kind of flexibility can be very useful, not least in the encoding and manipulation of infinite data structures. That said, we would also like to make it clear at this early stage that none of the extensions to Haskell that we put forward in this thesis relies on the semantics being non-strict. Thus it is perfectly reasonable to judge the merits of our extensions as if they were intended for an ordinary, strict programming language.

Function definitions

Functions can be defined by equations on the top-level of a program, or locally within a single expression, as in

```
f x y z = let sq i = i * i
          in sq x * sq y * sq z
```

Note that the single equality symbol denotes *definitional* equality in Haskell (i.e. = is neither a destructive assignment, nor an equality test). Local definitions within other definitions are also possible, as in

```
f x y z = sq x * sq y * sq z where
   sq v = v * v
```

Anonymous functions can furthermore be introduced, with the so-called *lambda*-expression

```
\x y z -> x*y*z
```

being identical to

```
let h x y z = x*y*z in h
```

Type inference

In general, the programmer does not have to supply a type when introducing a new variable. Instead the Hindley-Milner-style type inference algorithm employed in Haskell is able to find the most general type for each expression, which often results in *polymorphic types* being inferred (i.e. type expressions that include variables standing for arbitrary types). The obvious example of a polymorphic type is given by the identity function,

```
id x = x
```

which has the most general type `a -> a` in Haskell. However, should the programmer so decide, an explicit type annotation can also be used to indicate a more specific type, as in

```
iid :: Int -> Int
iid x = x
```

Partial application

A function like `f` above that takes three integer arguments and delivers an integer result has the type

```
Int -> Int -> Int -> Int
```

However, this does not mean that such a function must always appear before exactly three arguments in an expression. Instead, a function applied to fewer arguments is treated as an expression denoting an anonymous function, which in turn is applicable to the missing arguments. This means that `(f 7)` is a valid expression of type `Int -> Int -> Int`, and that `(f 7 13)` denotes a function of type `Int -> Int`. Note that this treatment is consistent with parsing an expression like `f 7 13 0` as `(((f 7) 13) 0)`.

Pattern-matching

Haskell functions are often defined by a sequence of equations that *pattern-match* on their arguments, like in the following example:

```
fac 0 = 1
fac n = n * fac (n-1)
```

An equivalent, but arguably less elegant definition of the same function would be

```
fac n = if n==0 then 1 else n * fac (n-1)
```

Notice also the use of recursion in these two definitions, which is the prescribed (and only!) way of expressing iteration in a purely functional language.[1]

[1] Later in this chapter we will see examples of more traditional *loop* constructs being defined for monadic commands. It should be noted, though, that the semantics of these constructs are given in terms of recursion (and that an optimizing compiler, in turn, is likely to implement many forms of recursion in terms of loops on the machine language level).

Another form of pattern-matching, using Boolean *guard* expressions, can also be used, although this form might appear a bit contrived in this simple example.

```
fac n | n==0      = 1
      | otherwise = n * fac (n-1)
```

Moreover, explicit **case** expressions are also available in Haskell, as in this fourth variant of the factorial function:

```
fac n = case n of
           0 -> 1
           m -> m * fac (m-1)
```

Algebraic datatypes

User-defined types take the form of *algebraic datatypes* in Haskell, which is a kind of labeled union types with name equality and recursive scope. Here is an example of a type for binary trees:

```
data BTree a = Leaf a
             | Node (BTree a) (BTree a)
```

The type argument `a` is used to make the binary tree type polymorphic in the contents of its leaves; thus a binary tree of integers has the type `BTree Int`. The identifiers `Leaf` and `Node` implicitly defined above are called the *constructors* of the datatype. Constructors, which have global scope in Haskell, can be used both as function identifiers and in patterns, as the following example illustrates:

```
swap v@(Leaf _) = v
swap (Node l r) = Node (swap r) (swap l)
```

This function (of type `BTree a -> BTree a`) takes any binary tree and returns a mirror image of the tree obtained by recursively swapping its left and right branches. The first defining equation also illustrates the use of two special Haskell patterns: the *as*-pattern `v@` ... which binds an additional variable to a pattern, and the wildcard `_` which matches anything without binding any variables.

Predefined types

In addition to the integers, Haskell's primitive types include characters (`Char`) as well as floating-point numbers (`Float` and `Double`). The type of Boolean values (`Bool`) is a predefined, but still ordinary algebraic datatype.

Lists and tuples are also essentially predefined datatypes, but they are supported by some special syntax. The empty list is written `[]`, and a non-empty list with head `x` and tail `xs` is written `x:xs`. A list known in its entirety can be expressed as `[x1,x2,x3]`, or equivalently `x1:x2:x3:[]`. Moreover, a pair of elements `a` and `b` is written `(a,b)`, and a triple also containing `c` is written

`(a,b,c)`, etc. As an illustration to these issues, here follows a function which "zips" two lists into a list of pairs:

```
zip (a:as) (b:bs) = (a,b) : zip as bs
zip _      _      = []
```

The type names for lists and tuples are formed in accordance with the term syntax: `[a]` is the type of lists containing elements of type `a`, and `(a,b)` denotes the type of pairs formed by elements of types `a` and `b`. Thus the type of the function `zip` above is `[a] -> [b] -> [(a,b)]`. There is also degenerate tuple type `()`, called *unit*, which only contains the single element `()`.

Strings are just lists of characters in Haskell, although for this specific type ordinary string syntax can also be used, with `"abc"` being equivalent to `['a','b','c']`. The type name `String` is just a *type abbreviation*, defined as:

```
type String = [Char]
```

String concatenation is thus an instance of general list concatenation in Haskell, for which there exists a standard operator `++`, defined as

```
[]     ++ bs = bs
(a:as) ++ bs = a : (as ++ bs)
```

Haskell also provides a primitive type `Array`, with an indexing operator `!` and an "update" operator `//`. However, this type suffers from the fact that updates must be implemented in the purely functional way, which often amounts creating fresh copies of an array each time it is modified. We will see later in this chapter how monads and stateful objects may enable us to support the `Array` type in a more intuitive, as well as a more efficient manner.

Higher-order functions

Functions are first-class values in Haskell, so it is quite common that functions are defined to take other functions as parameters. A typical example of this is the standard function `map`, which maps a given list into a new list by applying some unspecified function to each element. `map` is defined as follows:

```
map f []     = []
map f (x:xs) = f x : map f x
```

The higher-orderedness of `map` is exposed in its type,

```
map :: (a -> b) -> [a] -> [b]
```

where it should be noted that the parentheses are essential. As an example of how `map` can be used, we construct a capitalizing function for strings by defining

```
cap = map toUpper
```

where `toUpper :: Char -> Char` is a predefined function that capitalizes characters. The type of `cap` must accordingly be

```
cap :: [Char] -> [Char]
```

or, equivalently,

```
cap :: String -> String
```

Layout

Haskell makes extensive use of two-dimensional layout in order to convey information that would otherwise have to be supplied using delimiter symbols. The intended meaning of this convention should hopefully be obvious, and we will continue to use it throughout in our examples. It is occasionally convenient to override the layout rules with a more explicit syntax, though, and hence it may be good to keep in mind that the following two-dimensional code fragment

```
let f x y = e1
    g i j = e2
in g
```

is actually a syntactic shorthand for

```
let { f x y = e1 ; g i j = e2 } in g
```

2.2 Records

The first O'Haskell contribution we encounter in this survey is a system for programming with first-class records. Although Haskell already provides some support for records, we have chosen to replace this feature with the present system, partly because the Haskell proposal is a somewhat ad-hoc adaption of the datatype syntax, and partly because Haskell records do not fit too well with the subtyping extension we will describe in the next section.

The distinguishing feature of our record proposal is that the treatment of records and datatypes is perfectly *symmetric*; that is, there is a close correspondence between record selectors and datatype constructors, between record construction and datatype selection (i.e. pattern-matching over constructors), and between the corresponding forms of type extension, which yields subtypes for records and supertypes for datatypes.

Along this line, we treat both record selectors and datatype constructors as *global* constants — an absolutely normal choice for what datatypes are concerned, but not so for records (see e.g. [MTH90, Gas97]). Still, we think that a symmetric treatment like the present one has some interesting merits in itself, and that the ability to form hierarchies of record types alleviates most of the problems of having a common scope for all selector names. We also note that overloaded names in Haskell are given very much the same treatment, without much hassle in practice.

A record type is defined in O'Haskell by a global declaration on par with the datatype declarations previously encountered. The following example shows a

record type describing two-dimensional points, defined by two selector identifiers of type `Float`.

```
struct Point =
   x,y :: Float
```

The **struct** keyword is reused in the term syntax for record construction. We will generally rely on Haskell's layout-rule to avoid cluttering up our record expressions, as in the following example:

```
pt = struct
        x = 0.0
        y = 0.0
```

Since the selector identifiers are global and unique, there is no need to indicate which record type a record term belongs to. It is a static error to construct a record term where the set of selector equations is not exhaustive for some record type.

Record selection is performed by means of the standard *dot*-syntax. O'Haskell distinguishes record selection from the predefined Haskell operator `.` by requiring that the latter expression is followed by some amount of white space before any subsequent identifier. Record selection also binds more tightly than function and operator application, as the following example indicates.

```
dist p = sqrt (sq p.x + sq p.y) where
  sq i = i * i
```

A selector can moreover be turned into an ordinary prefix function if needed, by enclosing it in parentheses, as in

```
xs = map (.x) some_list_of_points
```

Just as algebraic datatypes may take type arguments, so may record types. The following example shows a record type that captures the signatures of the standard equality operators.[2]

```
struct Eq a =
   eq, ne :: a -> a -> Bool
```

A record term of type `Eq Point` is defined below.

```
pdict = struct
           eq     = eq
           ne a b = not (eq a b)
        where
           eq a b = a.x==b.x && a.y==b.y
```

[2]Strictly speaking, this record type is not legal since its name coincides with that of a predefined Haskell *type class*. Type classes form the basis of the *overloading* system of Haskell, whose ins and outs are beyond the scope of this survey. The name `Eq` has a deliberate purpose, though — it makes the example connect on to a known Haskell concept, and in addition it indicates the potential possibility of reducing the number of constructs in O'Haskell by eliminating type class declarations in favour of record types. See further the section on future work in chapter 8.

This example also illustrates three minor points about records: (1) record expressions are not recursive, (2) record selectors possess their own namespace (the equation `eq = eq` above is *not* recursive), and (3) selectors may be implemented using the familiar function definition syntax if so desired.

2.3 Subtyping

The subtyping system of O'Haskell is based on *name inequality*. This means that a possible subtype relationship between (say) two record types is determined solely by the names of the involved types, and not by consideration to whether the record types in question might have matching definitions in some sense. Thus, name inequality is just a logical generalization of the name *equality* principle used in Haskell for determining whether two types are equal.

The subtype relation between user-supplied types is defined by explicit declaration. This makes record subtyping in O'Haskell look quite similar to interface extension in Java, as the following type declaration exemplifies:

```
struct CPoint < Point =
   color :: Color
```

The record type `CPoint` is here both introduced, and declared to be a subtype of `Point` by means of the *subtype axiom* `CPoint < Point`. A consequence of this axiom is that the type `CPoint` is considered to possess the selectors `x` and `y` as well, in addition to its own contributed selector `color`. This must be observed when constructing `CPoint` terms, as is done in the following function:

```
addColor p = struct x = p.x; y = p.y; color = Black

cpt = addColor pt
```

Notice here that leaving out the equation `color = Black` would not make the definition invalid, since the function result would then be a value of type `Point` instead of `CPoint`. On the other hand, the selectors `x` and `y` are vital, because without them the record term would not exhaustively implement any record type.

Subtyping can also be defined for algebraic datatypes. Consider the following type modeling the black and white colors:

```
data BW = Black
        | White
```

This type can now be used as the basis for an extended color type:

```
data Color > BW =
   Red | Orange | Yellow | Green | Blue | Violet
```

Since its set of possible values is larger, the new type `Color` defined here must necessarily be a *supertype* of `BW` (hence we use the symbol `>` instead of `<` when

extending a datatype). The subtype axiom introduced by the previous type declaration is accordingly `BW < Color`. And analogously to the case for record types formed by extension, the extended type `Color` is considered to possess all the constructors of its base type `BW`, in addition to those explicitly mentioned for `Color`.

Haskell allows pattern-matching to be incomplete, so there is no datatype counterpart to the static exhaustiveness requirement that exists for record types. However, the set of constructors associated with each datatype still influences the static analysis of O'Haskell programs, in the sense that the type inference algorithm approximates the domain of a pattern-matching construct to the smallest such set that contains all enumerated constructors. The two following functions, whose domains become `BW` and `Color`, respectively, illustrate this point.

```
f Black = 0
f _     = 1

g Black = 0
g Red   = 1
g _     = 2
```

2.3.1 Polymorphic subtype axioms

Subtype axioms may be polymorphic, as in the following example where a record type capturing the standard set of comparison operators is formed by extending the type `Eq` defined in a previous section.

```
struct Ord a < Eq a =
   lt, le, ge, gt :: a -> a -> Bool
```

The subtype axiom declared here states that for all types `a`, a value of type `Ord a` also supports the operations of `Eq a`.

Polymorphic subtyping works just as well for datatypes. Consider the following example, which provides an alternative definition of the standard Haskell type `Either`.

```
data Left a =
   Left a

data Right a =
   Right a

data Either a b > Left a, Right b
```

Apart from showing that a datatype declaration need not necessarily declare any new value constructors, the last type declaration above is also an example of type extension with multiple basetypes. It effectively introduces *two* polymorphic subtype axioms; one which says that for all `a` and `b`, a value in `Left a` also

belongs to `Either a b`, and one which reads: for all `a` and `b`, the values of type `Right b` form a subset of the values of type `Either a b`.

So, for some fixed `a`, a value of type `Left a` can also be considered to be of type `Either a b`, *for all* `b`. This typing flexibility is actually a form of rank-2 polymorphism [Lev83], which is put to good use in the following example.

```
f v@(Left _) = v
f (Right 0)  = Right False
f (Right _)  = Right True
```

The interesting part here is the first defining equation. Thanks to subtyping, `v` becomes bound to a value of type `Left a` instead of `Either a Int`. Hence `v` can be reused on the right-hand side, in a context where a value of some supertype to `Right Bool` is expected. The O'Haskell type of `f` thus becomes

```
f :: Either a Int -> Either a Bool
```

Notice also that although both syntactically and operationally a valid Haskell function, `f` is not typeable under Haskell's type regime.

2.3.2 Depth subtyping

Subtyping is a reflexive and transitive relation; i.e. we have that any type is a subtype of itself, and that `S < T` and `T < U` implies `S < U` for all types `S`, `T`, and `U`. The fact that type constructors may be parameterized makes these issues a bit more complicated, though. For example, under what circumstances should we be able to conclude that `Eq S` is a subtype of `Eq T`?

O'Haskell incorporates a quite flexible rule that allows *depth subtyping* within a type constructor application, by taking the *variance* of a type constructor's parameters into account. By variance we mean the role a type variable has in the set of type expressions in its scope – does it occur in a function argument position, in a result position, in both these positions, or perhaps not at all?

In the definition of record type `Eq` above, all occurrences of the parameter `a` are to the left of a function arrow. For these cases O'Haskell prescribes *contravariant* subtyping, which means that `Eq S` is a subtype of `Eq T` only if `T` is a subtype of `S`. Thus we have that `Eq Point` is a subtype of `Eq CPoint`, i.e. an equality test developed for points can also be used for partitioning colored points into equivalence classes.

The parameter of the datatype `Left`, on the other hand, only occurs as a top-level type expression (that is, in a result position). In this case subtyping is *covariant*, which means for example that `Left CPoint` is a subtype of `Left Point`.

As an example of *invariant* subtyping, consider the record type

```
struct Box a =
   in  :: a -> Box a
   out :: a
```

Here the type parameter `a` takes the role of a function argument as well as a result, so both the co- and contravariant rules apply at the same time. The net result is that `Box S` is a subtype of `Box T` only if `S` and `T` are identical types.

There is also the unlikely case where a parameter is not used at all in the definition of a record or datatype:

```
data Contrived a = Unit
```

Clearly a value of type `Contrived S` also has the type `Contrived T` for any choice of `S` and `T`, thus depth subtyping for this *nonvariant* type constructor can be allowed without any further preconditions.

The motivation behind these rules is of course the classical rule for depth subtyping at the function type, which says that `S -> T` is a subtype of `S' -> T'` only if `S'` is a subtype of `S`, and `T` is a subtype of `T'` [CW85]. O'Haskell naturally supports this rule, as well as covariant subtyping for the built-in container types lists, tuples, and arrays.

Depth subtyping may now be transitively combined with subtype axioms to infer intuitively correct, but perhaps not immediately obvious subtype relationships. Some illustrative examples are:

Relation:	Interpretation:
`Left CPoint < Either Point Int`	*If either some kind of point or some integer is expected, a colored point will certainly do.*
`Ord Point < Eq CPoint`	*If an equivalence test for colored points is expected, a complete set of comparison operations for arbitrary points definitely meets the goal.*

2.3.3 Restrictions on subtype axioms

The general rule when defining subtype axioms is that a defined sub-/supertype (a *base* type) may be any kind of type expression that is not a variable. This means for example that arguments to parameterized base types may be constant type expressions if so desired, they are not limited to just variables.

This general rule is not without restrictions, though. The following examples illustrate the kind of subtype axioms that O'Haskell is forced to reject:

struct `S a < Bool`	(Supertype is not a record type)
struct `S a < S Int`	(Axiom interferes with depth subtyping)
struct `S a < Eq Char, Eq Int`	(Axioms are ambiguous)
struct `S a < Eq (S a)`	(Axiom is recursive)

All but the last of these restrictions are motivated by the type soundness criterion — allowing such axioms easily leads to typings that would break type safety, or at least require yet unknown implementation techniques for both records and datatypes. Recursive axioms, on the other hand, are probably operationally sound, but the current inference algorithm is unable to handle them. Chapter 5 will develop these technical issues in more detail.

The above restrictions also apply to the implicit subtyping relationships that can be derived by transitivity. Thus, assuming that we have

```
struct Num a < Eq a
    ...
```

the declaration

```
struct A < Ord Char, Num Int
```

is illegal, since its axioms also implicitly declare that `A < Eq Char` and `A < Eq Int`. Note, though, that a declaration like

```
struct A < Ord Char, Num Char
```

is OK, since all transitively generated rules that relate `A` to `Eq` now collapse into one: `A < Eq Char`.

A corresponding set of restrictions exists for subtype axioms relating algebraic datatypes. A formal definition of these restrictions will appear in chapter 5.

2.4 Automatic type inference

As is well known, polymorphic subtyping systems need types qualified by *subtype constraints* in order to preserve a notion of principal types. This is easily demonstrated by the following archetypical polymorphic function:

```
twice f x = f (f x)
```

In Haskell, `twice` has the principal type

```
(a -> a) -> a -> a
```

from which every other valid type for `twice` (e.g. `(Point -> Point) -> Point -> Point`) can be obtained as a substitution instance. But if we now allow subtyping, and assume `CPoint < Point`, `twice` can also have the type

```
(Point -> CPoint) -> Point -> CPoint
```

which is not an instance of the principal Haskell type. In fact, there can be no simple type for `twice` that has both `(Point -> Point) -> Point -> Point` and `(Point -> CPoint) -> Point -> CPoint` as substitution instances, since the greatest common anti-instance of these types, `(a -> b) -> a -> b`, is not a valid type for `twice`. So to obtain a notion of principality in this case, we must

restrict the possible instances of `a` and `b` to those types that allow a subtyping step from `b` to `a`; that is, to associate the subtype constraint `b < a` with the typing of `twice`. In O'Haskell subtype constraints are attached to types by means of the symbol `|`[3], so the principal type for `twice` thus becomes

```
(a -> b) -> a -> b | b < a
```

This type has two major drawbacks compared to the principal Haskell type: (1) it is syntactically longer than most of its useful instances because of the subtype constraint, and (2) it is no longer unique modulo renaming, since it can be shown that, for example,

```
(a -> b) -> c -> d | b < a, c < a, b < d
```

is also a principal type for `twice`. In this simple example the added complexity that results from these drawbacks is of course manageable, but even just slightly more involved examples soon get out of hand, since, in effect, *every application node* in the abstract syntax tree can give rise to a new type variable and a new subtype constraint. Known complete inference algorithms tend to illustrate this point very well, and even if simplification algorithms have been proposed that alleviate the problem to some extent, the general simplification problem is at least NP-hard. Apart from that, it is also an inevitable fact that no conservative simplification strategies in the world can ever give us back the attractive type for `twice` we know from Haskell.

For these reasons, O'Haskell relinquishes the goal of complete type inference, and employs a *partial* type inference algorithm that trades in generality for a consistently readable output. The basic idea of this approach is to let functions like `twice` retain their original Haskell type, and, in the spirit of monomorphic object-oriented languages, infer subtyping steps only when both the inferred and the expected type of an expression are known. This choice can be justified on the grounds that `(a -> a) -> a -> a` is still likely to be a sufficiently general type for `twice` in most situations, and that the benefit of a consistently readable output from the inference algorithm will arguably outweigh the inconvenience of having to supply a type annotation when this is not the case. We certainly do not want to prohibit exploration of the more elaborate areas of polymorphic subtyping that need constraints, but considering the cost involved, we think it is reasonable to expect the programmer to supply the type information in these cases.

As an example of where the lack of inferred subtype constraints might seem more unfortunate than in the typing of `twice`, consider the function

```
min x y = if less x y then x else y
```

which, assuming `less` is a relation on type `Point`, will be assigned the type

```
Point -> Point -> Point
```

by our algorithm. A more useful choice would probably have been

[3]The syntax is inspired by the way patterns with Boolean guards are expressed in Haskell.

```
a -> a -> a | a < Point
```

here, but as we have indicated, such a constrained type can only be attained by means of an explicit type annotation in O'Haskell. On the other hand, note that the principal type for `min`,

```
a -> b -> c | a < Point, b < Point, a < c, b < c
```

is a yet more complicated type, and presumably an overkill in any realistic context.

An informal characterization of our inference algorithm is that it improves on ordinary polymorphic type inference by allowing subtyping steps at application nodes when the types are known, as in

```
addColor cpt
```

for example. In addition, the algorithm computes least upper bounds for instantiation variables when required, so that e.g. the list

```
[cpt, pt]
```

will receive the type

```
[Point]
```

Greatest lower bounds for function arguments will also be found, resulting in the inferred type

```
CPoint -> (Int,Bool)
```

for the term

```
\p -> (p.x, p.color == Black)
```

Notice, though, that the algorithm assigns constraint-free types to *all* subterms of an expression, hence a compound expression might receive a less general type, even though its principal type has no constraints. One example of this is

```
let twice f x = f (f x) in twice addColor pt
```

which is assigned the type `Point`, not the principal type `CPoint`.

Unfortunately, a declarative specification of the set of programs that are amenable to this kind of partial type inference is still an open problem. Completeness relative to a system that lacks constraints is also not a realistic property to strive for, due to the absence of principal types in such a system. However, experience strongly suggests that the algorithm is able to find solutions to most constraint-free typing problems that occur in practice — in fact, an example of where it mistakenly fails has yet to be found in practical O'Haskell programming. Moreover, the algorithm is provably complete w.r.t. the Haskell type system, hence it possesses another very important property: programs typeable in Haskell retain their inferred types when considered as O'Haskell programs. On top of this, the algorithm can also be shown to accept all programs typeable in the core type system of Java.

2.5 Reactive objects

The central dynamic notion in O'Haskell is the state-encapsulating, concurrently executing *reactive object*. In this section we will survey this dynamic part of the language as it is seen by the programmer. However, the amount of new concepts introduced here is quite extensive, and some of these might not seem immediately compatible with the idea of a purely functional language. Thus some readers might find it helpful to know right from the outset that all constructs introduced in this section are indeed syntactically transformable into a language consisting of only the Haskell kernel and a set of primitive monadic constants. This "naked" view of O'Haskell will not be further pursued, though, until we begin our formal treatment of the subject in chapter 6.

2.5.1 Templates and methods

Objects are instantiated from a **template** construct, which defines the *initial state* of an object together with a *communication interface*. A communication interface can be a value of any type, but in order to be useful it must contain at least one *method* that will allow the object to react to method invocations (we will also use the metaphor *sending a message* as a synonym for invoking a method).

A method in turn can be of two forms: either an asynchronous **action**, that lets an invoking sender continue immediately and thus introduces concurrency, or a synchronous **request**, which allows a value to be passed back to the waiting sender. The body of a method, finally, is a sequence of *commands*, which can basically do three things: update the local state, create new objects, and invoke methods of other objects.

The following code fragment defines a template for a simple counter object:

```
counter = template
             val := 0
          in struct
                inc  = action
                          val := val + 1
                read = request
                          return val
```

Instantiating this template creates a new object with its own unique state variable `val`, and returns an interface that is a record containing two methods: one asynchronous action `inc`, and one synchronous request `read`. Invoking `inc` means sending an asynchronous message to the counter object behind the interface, which will respond by updating its state. Invoking `read`, in turn, will perform a rendezvous with the counter, and return its current value.

2.5.2 Procedures and commands

Actions, requests and templates are all examples of expressions that denote commands in the monad `Cmd`. `Cmd` is actually a type constructor, with `Cmd a` denoting the type of reactive commands that may perform side-effects before returning a value of type `a`. The `Cmd` type is visible in the type of `counter`,

```
counter :: Cmd Counter
```

where `Counter` is assumed to be a record type defined as

```
struct Counter =
   inc  :: Cmd ()
   read :: Cmd Int
```

The result returned from the execution of a monadic command may be bound to a variable by means of the *generator* notation. For example

```
do c <- counter
   ...
```

means that the command `counter` is executed (i.e. the counter template is instantiated), and `c` becomes bound to the value returned (the interface of the new counter object).

Executing a command and discarding its result is simply written without the left-arrow. For example, the result of invoking an asynchronous method is always the uninteresting value `()`, so a suitable way of incrementing counter `c` is therefore

```
do c <- counter
   c.inc
```

The **do**-construct above is by itself a full-blown expression, representing a command sequence, or an anonymous *procedure*, that *when executed* first creates an object instance of the template `counter`, and then invokes the method `inc` of this object. The value returned by a procedure is the value returned by its last command, so the type of the above expression thus becomes `Cmd ()`.

Since the `read` method of `c` is a synchronous method that returns an `Int`, we can write

```
do c <- counter
   c.inc
   c.read
```

and obtain a procedure with the type `Cmd Int`.

Just as for other commands, the result of executing the `read` method can be captured by means of the generator notation:

```
do c <- counter
   c.inc
   v <- c.read
   return v
```

This procedure is actually equivalent to the previous one. The identifier `return` is the trivial built-in command which produces a result value (in this case simply `v`) without performing any effects. Unlike most imperative languages, however, `return` is not a branching construct in O'Haskell — a `return` in the middle of a command sequence means just that a command without effects is executed and its result is discarded. For example

```
do ...
   return (v+1)
   return v
```

is simply identical to

```
do ...
   return v
```

Naming a procedure is moreover just like naming any other expression:

```
testCounter = do c <- counter
                 c.inc
                 c.read
```

`testCounter` is thus the name of a simple procedure which *whenever executed* creates a new counter object and returns its value after it has been incremented once. The counter itself is then simply forgotten, which means that its space requirements eventually will be reclaimed by the garbage collector.

A very useful procedure with a predefined name is

```
done = do return ()
```

which takes the role of a *null* command in O'Haskell.

2.5.3 A word about overloading

Sequencing by means of the **do**-construct, and command injection (via `return`), is not just limited to the `Cmd` monad. Indeed, just as in Haskell, these fundamental operations are *overloaded* and available for any type constructor that is an instance of the *type class* `Monad` [Jon93, HHJW94]. Type classes and the overloading system will not be covered in this thesis, though, partly because this feature constitutes a virtually orthogonal complement to the subtyping system we put forward, and partly because we do not intend to capitalize on overloading in any essential way in our presentation. In particular, monadic programming in general will not be a topic of this thesis.

Still, one more monad will be introduced, that is related to `Cmd` by means of subtyping. We will therefore take the liberty of reusing `return` and the **do**-syntax for this new type constructor, even though strictly speaking this means that the overloading system must come into play behind the scenes. And while we are on the subject, the same trick is actually tacitly employed for the equality operator `==` at a few places as well. However, the overloadings that occur in

this thesis are all statically resolvable, so our naive presentation of the matter is intuitively quite correct. It is our intention that this slight toning down of Haskell's most conspicuous type system feature will avoid far more confusion about overloading in O'Haskell than it gives rise to.

2.5.4 Assignable local state

The method-forming constructs **action** and **request** are syntactically similar to procedures, but with different operational behaviours. Whereas calling a procedure means that its commands are executed by the calling thread, invoking a method triggers execution of commands within the object to which the method belongs. For this reason, methods have no meaning in isolation, and the method syntax is accordingly not available outside the scope of a template.

In actions and requests, as well as in procedures that occur within the scope of a template, two additional forms of commands are available: commands that refer to local state variables (for example the command `return val` in the counter template), and commands that assign new values to these variables (e.g. `val := val + 1`).

As can be expected, state variables are surrounded by several restrictions in order to preserve purely functional semantics of expression evaluation. Firstly they must only occur within commands. For example

```
template
   x := 1
in x
```

is statically illegal, since the state variable `x` is not visible outside the commands of a method or a procedure. Secondly, there are no aliases in O'Haskell, which means that state variables are not first-class values. Thus the procedure declaration

```
someProc r = do r := 1
```

is illegal even if only applied to integer-typed state variables, because `r` is not syntactically a state variable. Parameterization over some unknown state can instead be achieved in O'Haskell by turning the possible parameter candidates into full-fledged objects.

Thirdly, the visibility of a state variable does not reach beyond the innermost enclosing template. This makes the following example ill-formed:

```
template
   x := 1
in
   template
      y := 2
   in
      do x := 0
```

And fourthly, there is a restriction which prevents other local bindings from shadowing a state variable. An expression like the following one is thus disallowed:

```
template
   x := 1
in \x -> ...
```

While not strictly necessary for preserving the purity of the language, this last restriction has the merit of making the question of assignability a simple syntactical matter (see section 6.1), at the same time as it puts the focus on the very special status that state variables enjoy in O'Haskell.

2.5.5 The `O` monad

Commands that refer or assign to the local state of an object belong to a richer monad `O s`, where the `s` component captures the type of the local state, and where `O s a` accordingly is the type of state-sensitive commands that return results of type `a`. An assignment command always returns `()`, whereas a state-referencing command can return any type. Due to subtyping (more on that below), procedures that contain at least one such command also become lifted to the monad `O s`.

The local state type of an object with more than one state variable is an anonymous tuple type; i.e. there is no information regarding the name of state variables encoded in the state type. For example, the templates

```
a = template
       x := 1
       f := True
    in ...
```

and

```
b = template
       count := 0
       enable := False
    in ...
```

both generate objects with local states of type `(Int,Bool)`. In fact, the anonymity of state variables effectively illustrates that procedures are *parametric* in the actual state they work on — there exists no connection at run-time between a value of some `O` type and the object in which it is declared. Thus, as long as the state types match, a procedure declared within one template may very well work as a local procedure within another template. This particularly means that the possibility of exporting state-dependent procedures does not constitute a loophole in the encapsulation mechanism that O'Haskell provides — still the only way by which an object may affect the state of another object is by invoking an exported *method*.

2.5.6 Self

Closely connected to the concept of a local state is the special variable `self`, which implicitly is in scope inside every template expression, and which may not be shadowed. All occurrences of `self` have type `Ref s`, where `s` is the type of the current local state. The values of type `Ref s` are the actual object references that uniquely identify a particular object at run-time. References are thus very similar to the *PIDs* (process identifiers) that most operating systems generate; they can also be thought of as a form of *pointers* to storage of type `s`. However, since actions and requests implicitly refer to the reference value bound to `self`, most programs do not need to explicitly access this variable. Some further details concerning the `Ref` type are provided in section 2.5.8, although the full significance of `self` will not become clear until the object semantics has been formalized in chapter 6.

It should also be noted that the variable `self` in O'Haskell has nothing to do with the *interface* of an object (in contrast to, for example, `this` in C++ and Java). This is a natural consequence of the fact that an O'Haskell object may have multiple interfaces — some objects may even generate new interfaces on demand (recall that an interface is simply a value that contains at least one method).

2.5.7 Expressions vs. commands

Although commands are full-fledged values in O'Haskell, there is a sharp distinction between the *execution* of a command, and the *evaluation* of a command considered as a functional value. The following examples illustrate the point.

```
f :: Counter -> (Cmd (), Cmd Int)
f cnt = (cnt.inc, cnt.read)
```

The identifier `f` defined here is a function, not a procedure; it cannot be *executed* in any sense, only applied to arguments of type `Counter`. The fact that the returned pair has command-valued components does not change the status of `f`. In particular, the occurrence of subexpressions `cnt.inc` and `cnt.read` in the right-hand side of `f` does *not* imply that the methods of some counter object get invoked when evaluating applications of `f`.

Extracting the first component of a pair returned by `f` is also a pure evaluation with no side-effects. However, the result in this case is a command value, which has the specific property of being *executable*. By placing a command value in the the command sequence of a procedure, the command becomes subject to execution, *whenever the procedure itself is called*. Such a procedure is shown below.

```
do c <- counter
   fst (f c)
```

The separation between *evaluation* and *execution* of command values can be made more explicit by introducing a name for the evaluated command. This

is achieved by the **let**-command, which is a purely declarative construct (the equality sign really denotes equality).

```
do c <- counter
   let a = fst (f c)
   a
```

Hence the two preceding examples are actually equivalent, and the counter created will be incremented just once. The following fragment is yet another equivalent example,

```
do c <- counter
   let a = fst (f c)
       b = fst (f c)
   a
```

whereas the next procedure has a different operational behaviour (here the `inc` method of `c` will actually be invoked twice).

```
do c <- counter
   let a = fst (f c)
   a
   a
```

A computation that behaves like `f` above, but which also as a side-effect increments the counter it receives as an argument, must be expressed as a procedure.

```
g :: Counter -> Cmd (Cmd (), Cmd Int)
g cnt = do c.inc
           return (c.inc, c.read)
```

Note that the type system clearly separates the side-effecting computation from the pure one, by applying the `Cmd` type constructor to the range type if the computation happens to be a procedure.

Likewise, the type system demands that computations which depend on the current state of some object be implemented as procedures. For example,

```
h :: Counter -> Int
h cnt = cnt.read * 10
```

is not type correct, since `cnt.read` is not an integer — it is a *command* which *when executed* returns an integer. If we really want to compute the result of multiplying the counter value with 10 we must write

```
h :: Counter -> Cmd Int
h cnt = do v <- cnt.read
           return (v * 10)
```

2.5.8 Subtyping in the `O` monad

We have already indicated that the `Cmd` and `O s` monads are related by subtyping. This is formally expressed as a built-in subtype axiom:

```
Cmd a < O s a
```

This axiom can preferably be read as a higher-order relation: "all commands in the monad `Cmd` are also commands in the monad `O s`, for any `s`". Note also that the definition above is another example of the rank-2 polymorphism made possible by a subtype relation based on polymorphic subtype axioms.

One way of characterizing the `Cmd` monad is as a refinement of the `O` monad, that captures the set of all commands that are independent of the current local state. O'Haskell actually takes this idea even further, by providing three more primitive command types, that are related to the `Cmd` monad via the following built-in axioms:

```
Template a < Cmd a
Action     < Cmd ()
Request a  < Cmd a
```

The intention here is of course to provide even more precise typings of the **template**, **action**, and **request** constructs. Thus, the type inferred for the template `counter` defined at the beginning of this section is actually `Template Counter` (instead of `Cmd Counter`), and the types of its two methods are accordingly `Action` and `Request Int`. The record type `Counter` can of course be updated to take advantage of this increased precision:

```
struct Counter =
   inc  :: Action
   read :: Request Int
```

Unlike the refinement step of going from `O s` to the `Cmd` monad (which actually makes more programs typeable because of the rank-2 polymorphism), the distinction between `Cmd` and its subtypes has mostly a documentary value. However, by turning a documentation practice into a type system matter, the type system can additionally be relied on for guaranteeing certain operational properties. For example, a command of type `Template a` can be relied on not to change the state of any existing objects when executed, since a template instantiation only adds components to the system state. Moreover, commands of type `Action` or `Template a` are guaranteed to be deadlock-free, since a synchronous method can never possess any of these types. Note that none of these properties hold for a general command of type `Cmd a`.

To facilitate straightforward comparison of arbitrary object reference values, O'Haskell provides yet another primitive type with a built-in subtype axiom:

```
Ref a < UniRef
```

By means of this axiom, all object references can be compared for equality (by the overloaded primitive `==`) when considered as values of the supertype `UniRef`. O'Haskell moreover provides a predefined record type `Object` for this purpose, which forms a convenient base from which interface types supporting a notion of object identity can be built.

```
struct Object =
   self :: UniRef
```

Of the type constructors mentioned here, `Cmd`, `Template`, and `Request` are all covariant in their single argument. This also holds for the `O` type in case of its second argument, whereas the same constructor, like all types that support both dereferencing and assignment, must be invariant in its state component. Along this line, `Ref` is also forced to be an invariant type constructor.

2.5.9 The `main` procedure

Like in many other languages, the procedure name `main` is special in O'Haskell. This procedure, which must be defined in every program, is implicitly invoked whenever an O'Haskell program is run. The following example shows a program which calls the procedure `testCounter` to obtain an integer value, and then sends this result (converted to a string by `show`) to the standard printing routine of the computing environment.

```
main env = do v <- testCounter
              env.putStr (show v)
```

In contrast to Haskell, `main` takes the computing environment as an argument. This means that the possibility of performing external effects is controllable by the programmer, by appropriate use of parameterization. More details regarding the type of the environment parameter will be further discussed under the heading *Reactivity* below.

2.5.10 Concurrency

An object is an implicit critical section, that is, at most one of its methods can be active at any time. The following example creates a situation with two contending clients sending messages to a counter object. Mutual exclusion between method invocations guarantees that the classical problem of simultaneous updates to a state variable still does not exist.

```
proc cnt = template
              --
           in action
                 cnt.inc

main env = do c <- counter
              p <- proc c
```

```
p
c.inc
v <- c.read
env.putStr (show v)
```

Methods are furthermore guaranteed to be executed in the order they are invoked (although concurrent execution of multiple objects can make the actual invocation order nondeterministic). This is also illustrated by the previous example. In the `main` procedure it is safe to assume that the value `v` read back from the counter is at least 1, since even though `inc` is an asynchronous method, it must have been processed before the subsequent `read` method call returns. On the other hand, nothing can be said about whether the concurrently executing action of object `p` will be scheduled to make its `inc` call before, in between, or after the two invocations in procedure `main`.

2.5.11 Reactivity

Objects alternate between phases of indefinitely long inactivity and finite method execution. For many applications, the active phases may even be considered momentary, given a sufficiently fast processor. The existence of value-returning synchronous methods does not change that fact, since, assuming that the system is not in deadlock, there are no *other* commands that may block indefinitely, and hence invoking a request cannot do so either. Thus it is important that the computing environment also adheres to this reactive view, by *not* providing any indefinitely blocking operations.

Instead, interactive O'Haskell programs install *callback methods* in the computing environment, with the intention that they will be invoked whenever the event occurs that they are set to handle. As a consequence, O'Haskell programs do not generally terminate when the main routine returns; they are rather considered to be alive as long as there is at least one active object or one installed callback method in the system (or, alternatively, execution may terminate when an object invokes the `quit` method of the environment).

The actual shape of the interface to the computing environment must of course be allowed to vary with the type of application being constructed. The current O'Haskell implementation supports three different environment types, `TkEnv`, `XEnv`, and `StdEnv`, which model the computing environments offered by a Tk server, an X server, and the *stdio* fragment of a Unix operating system, respectively. Several other suggestively named interfaces types are of course also conceivable, e.g. `AppletEnv`, `COMEnv`, `SysEnv`, etc.

A skeleton for a program running under the `StdEnv` environment follows below.

```
prog env = template
               ...
           in struct
                  getChar c = action
                                  ...
```

```
                signal n  = action
                                ...

main env = do p <- prog env
              env.interactive p
```

We only show a minimal set of the operations offered by `StdEnv` here, but the main idea behind text-based, interactive programming in O'Haskell should still be evident.

```
struct StdEnv =
   putStr      :: String -> Action
   interactive :: StdProg -> Action
   quit        :: Action
   ...

struct StdProg =
    getChar :: Char -> Action
    signal  :: Int -> Action
```

The type `StdProg` groups the main event-handlers in a text-oriented program together in a common structure, which is installed in the environment in a single step. Note specifically that `getChar` is a method of the *application*, not the environment, in this reactive formulation. Also note that unless the interactive application performs excessively long computations in response to each input character, signal handling will be almost momentary.

As an example of a more elaborate environment interface, `TkEnv` will be discussed as part of a programming example in the next chapter.

2.6 Additional extensions

O'Haskell also provides a number of minor, mostly syntactic extensions to the Haskell base, which we will briefly review in this section.

2.6.1 Extended do-syntax

The **do**-syntax of Haskell already contains an example of an expression construct lifted to a corresponding role as a command: the **let**-command, illustrated in subsection 2.5.7. O'Haskell defines commands corresponding to the **if**- and **case**-expressions as well, using the following quite intuitive syntax:

```
do if e then
      cmds
   else
      cmds
   if e then
      cmds
```

```
   case e of
      p1 -> cmds
      p2 -> cmds
```

In addition, O'Haskell provides syntactic support for recursive generator bindings, iteration, and exception handling:

```
do fix x <- cmd y
       y <- cmd x
   forall i <- e do
      cmds
   while e do
      cmds
handle
   exception1 -> cmds
   exception2 -> cmds
```

The **handle**-clause is an optional appendix to the **do**-construct, with the implied semantics that it handles the listed exceptions within the whole proper command sequence. Like the rest of the **do**-construct, the translation of these syntactic forms just relies on the existence of some (fairly) standard monadic constants (see appendix A). The new command-forms are also overloaded in the full language, although the **while**-command is not of very much use outside the `O` monad, with its support for state variables.

2.6.2 Array updates

To simplify programming with the primitive `Array` type, O'Haskell supports a special array-update syntax for arrays declared as state variables. Assuming `a` is such an array, an update to `a` at index `i` with expression `e` can be done as follows (recall that the array indexing operator in Haskell is `!`):

```
a!i := e
```

Semantically, this form of assignment is equivalent to

```
a := a // [(i,e)]
```

where `//` is Haskell's pure array update operator. But apart from being intuitively simpler, the former syntax has the merit of making it clear that normal use of an encapsulated array is likely to be single-threaded; i.e. the rare cases where `a` is mentioned for another purpose than indexing become easily identifiable. Hence conservative copying of the array can be reserved for these occasions, and ordinary updates to `a` performed in place, which is also exactly what the refined syntax above suggests.

See further the discussion on implementation details in chapter 7.

2.6.3 Record stuffing

Record expressions may optionally be terminated by a type constructor name, as in the following examples:

```
struct ..S

struct a = exp; b = exp; ..S
```

These expressions utilize *record stuffing*, a syntactic device for completing record definitions with equations that just map a selector name to an identical variable already in scope. The missing selectors in such an expression are determined by the appended type constructor (which must stand for a record type), on condition that corresponding variables are defined in the enclosing scope. So if `S` is a (possibly parameterized) record type with selectors `a`, `b`, and `c`, the two record values above are actually

```
struct a = a; b = b; c = c
```

and

```
struct a = exp; b = exp; c = c
```

where `c`, and in the first case even `a` and `b`, must already be bound. Record stuffing is most useful in conjunction with **let**-expressions, as we will see in the subsequent chapter.

2.7 What about inheritance?

Object-oriented modeling is deeply associated with a *classification paradigm*, which attempts to explore behavioral relations between objects according to a biologically inspired *inheritance*-metaphor [Weg87]. One side of such a classification scheme clearly concerns purely syntactical similarities between object interfaces, an aspect of inheritance that is well provided for in O'Haskell in terms of its subtyping system.

Behavioural inheritance in general, though, is a much more complex issue. In most object-oriented languages, inheritance is approximated as reuse of method implementations. This allows a new class of objects (a `template` in O'Haskell terminology) to be incrementally defined on basis of old ones, just as if the methods of the old classes had been syntactically copied directly into the definition of the new class. It also means, though, that the binding of method names to implementations cannot in general be statically known, but must be handled at runtime through a mechanism termed *dynamic binding*.

Dynamic binding can actually be understood as two separate issues:

1. Selection of code for an external method invocation, based not on the static type of the interface, but on the actual class associated with the object behind it (external dispatch).

2. A similar openness to the dynamic class even when the method call is originating from a *sibling method in the receptor object itself* (internal dispatch).

From a higher-order point of view, feature (1) is not very special; it comes for free once methods are elevated to first-class status. For example, an interface to an O'Haskell object of type `Counter` need not necessarily be generated by the previous template `counter`. An alternative origin could just as well be

```
counter2 = template
           in struct
              inc  = action
                        c.inc
                        c.inc
              read = request
                        c.read
```

where `c` might be an interface to a counter of the original kind.

Feature (2) on the other hand, is arguably the source of much of the semantic complexity associated with object-oriented languages. In O'Haskell terms it would mean that the template

```
counter3 = template
              val := 0
           in let
              inc  = action
                        val := val + 1
                        if val == 100 then doit
              read = request return val
              doit = done
           in struct ..ProtectedCounter

struct ProtectedCounter < Counter =
   doit :: Cmd ()
```

cannot be understood independently of its context, since its meaning must be kept sensitive to remote "redefinitions" of the exported procedure `doit`.

We have chosen not to support property (2) in O'Haskell, on basis of the following compelling chain of arguments [Wil90]:

- Most uses of dynamic binding in current object-oriented programs are examples of external dispatch.
- Of the remaining cases, all but a few are arguably concerned with overriding of empty (virtual) methods *that were foreseen to be overridden*. Such uses are better expressed by ordinary parameterization in O'Haskell, as in

```
counter4 doit = template
                   val := 0
                in struct
                   inc  = action
                             val := val + 1
                             if val == 100 then doit
                   read = request return val
```

- Of the remaining cases, all but a few are arguably overridings of default implementations of methods that were still foreseen to be overridden. This can also be captured in O'Haskell:

  ```
  counter5 = counter4 done
  ```

- Of the remaining cases, many are likely to be quick-and-dirty overridings that result in erroneous code. O'Haskell does not actively support these cases.
- Of the remaining cases, many are likely to depend on access to the full source code of the overridden methods. If source is available, corrections can be made in place. Alternatively, the code can be modified to support explicit parameterization as above.
- The cases that still remain are arguably too few to justify provision of a radically new semantics of recursive binding just for methods.

Hence it is justifiable to say that O'Haskell enforces a *closed-world* view of objects. This means that inheritance must become identical to simple *delegation*; i.e. the creation of private "parent" instances together with the task of plugging the right method values into the right interfaces. Overriding is likewise reduced to a matter of ordinary abstraction and application. We do however foresee that further experimentation with the language will identify the need for some special syntax to make this practice straightforward.

The question of inheritance in O'Haskell will be revistited in chapter 4, in the light of a concrete programming example.

Chapter 3

Examples

In this chapter we will present several programming problems coded in the reactive style of O'Haskell. The examples range from an interactive drawing program (section 3.1), over an embedded controller system (section 3.2), to a highly concurrent implementation of Sieve of Eratosthenes (section 3.3). We will also demonstrate reactive encodings of some classical blocking primitives in sections 3.4 to 3.6, as well as two slightly larger examples: a schematic implementation of a Telnet client (section 3.7), and a controller for a telephone exchange (section 3.8). Our most substantial example, a modular implementation of a self-stabilizing network protocol, will be presented separately in the form of a case study in chapter 4.

3.1 An interactive drawing program

In section 2.5.11 a reactive formulation of a simple character-based interactive program and its computing environment was outlined. Here we will refine the basic ideas of that example in several ways, by considering a program centered around graphics and mouse-based interaction. Our refined example is a small drawing program, *Scribbler*, that uses Ousterhout's GUI toolkit *Tk* as its computing environment [Ous94]. Scribbler is defined by the code that follows below.

```
struct Scribbler =
   penUp   :: Int -> Pos -> Action
   penDown :: Int -> Pos -> Action
   movePen :: Pos -> Action

scribbler  :: Canvas -> Template Scribbler
scribbler canv =
   template
      pos := Nothing
      col := Red
   in struct
```

```
      penUp   _   _ = action pos := Nothing
      penDown but p = action (pos,col) := (Just p,color but)
         where
            color 1 = Red
            color 2 = Green
            color _ = Blue
      movePen     p = action
         case pos of
            Nothing -> done
            Just p' -> canv.line p' p [Width 3, Fill col]
                       pos := Just p

main tk = do
   win   <- tk.window [Title "Scribbler"]
   canv  <- win.canvas [Background White]
   quit  <- win.button [Text "Quit", Command tk.quit]
   clear <- win.button [Text "Clear", Command canv.clear]
   s     <- scribbler canv
   win.pack (canv ^^^ (clear <<< quit))
   canv.bind [AnyButtonPress   s.penDown,
              AnyButtonRelease s.penUp,
              Motion           s.movePen]
```

The procedure `main` does a number of things in this example: it creates a window and a number of other graphical *widgets*, it creates the main object of the application, it arranges the widgets according to a specific layout (the `win.pack` method call), and it installs the desired event-handlers (by invoking `canv.bind`). Notice that in contrast to the `StdEnv` case the interface of the main object only plays an internal role here, since the multitude of events imaginable in a Tk environment makes it more realistic to install the desired event handlers one-by-one. By installing the handlers specifically with the canvas widget, it is understood that only events generated when the mouse is over the canvas will be reported.

The data constructors `Nothing` and `Just` used in this example are members of an important Haskell datatype, `Maybe`, that is defined as follows:

```
data Maybe a = Nothing | Just a
```

This type is useful in the case a value may be optionally absent, as the value of the current pen position can be in this drawing program. Notice that `Nothing` takes on the same role here as values like *nil* or *void* do in imperative languages, although in a robust and statically type safe manner.

The Tk environment

The O'Haskell interface to the Tk toolkit is interesting in its own right, since Tk is an example of a tried and tested graphical toolkit consciously designed

around an event-driven structure. The fact that Tk is originally untyped makes the task of giving it a flexible interface in O'Haskell even more challenging.

Every designer of a graphical toolkit is faced with the problem of how to express layout in a convenient manner, and how to make the parameters that control the appearance of the toolkit easily configurable. These problems are not trivial, since the naive solution of having the programmer supply coordinates as well as values for every configurable parameter makes programming with the toolkit an unduly exercise in details.

Tk addresses the first problem with a *packing algorithm* that is able to compute coordinates and sizes for widgets on basis of abstract information like whether stacking direction should be horizontal or vertical, or whether stretchable background "glue" should be inserted, etc. The packing algorithm is however quite complicated, and its interface has an unnecessarily imperative formulation, even for an algorithm written in an imperative language.

The latter problem above is solved in Tk by a calling convention that allows widget creating functions to take a variable number of configuration *options*, just like commands on the standard Unix command line. However, few statically typed languages would allow a direct adaption of this idea, and a translation that attempts to utilize subtyping is furthermore complicated by the fact that options for different widgets often overlap in a way that contradicts the intuitive relationships that one might want to express (e.g. that every button is a widget).

Our O'Haskell interface to Tk makes good use of polymorphism as well as both record and datatype subtyping to circumvent these problems. The basic widget operations are defined in a hierarchy of record types, where each widget type is parameterized w.r.t. the type of the configuration options it takes. We only show the significant parts of the type declarations here, and let incomplete declarations be denoted by trailing dots.

```
struct Widget a =
   focus     :: Action
   bind      :: [Event] -> Action
   config    :: [a] -> Action
   ...

struct WWidget a < Widget a, Packable

struct Button < WWidget ButtonOpt =
   flash     :: Action
   invoke    :: Action

struct Canvas < WWidget CanvasOpt =
   line      :: Pos -> Pos -> [LineOpt] -> Request Line
   clear     :: Action
   ...
```

```
struct CWidget a < Widget a =
   getCoords :: Request [Pos]
   setCoords :: [Pos] -> Action
   move      :: Pos -> Action

struct Line < CWidget LineOpt

struct Window < Widget WindowOpt =
   button    :: [ButtonOpt] -> Request Button
   canvas    :: [CanvasOpt] -> Request Canvas
   pack      :: Packable -> Action
   ...
```

This hierarchy particularly expresses that every widget subject to automatic layout is a subtype of the type `Packable`. Any object that has this type is a valid target for the packing algorithm of a top-level window. The various layout forms are expressed in terms of purely functional *layout combinators*, that hide the details of how the actual packer is invoked.

```
struct Packable =
   packIn    :: PathName -> Dir -> PFill -> Expansion -> Cmd ()

(<<<),(^^^)  :: Packable -> Packable -> Packable
                -- normal horizontal & vertical layout
(<*<),(^*^)  :: Packable -> Packable -> Packable
                -- layout + size expansion
...
```

The looks and behaviours of these combinators are actually a loan from the implementation of *TkGofer* [CMV97], although the idea of using combinators to define layout in a purely functional language is older (see e.g. [CH93]). The listed parameters of the `packIn` method are governed by Tk-specific details that are of little use to the programmer; the packing algorithm is instead initiated by invoking the `pack` method of some top-level window.

The various configuration options are defined as constructors in a hierarchy of datatypes. Since the widget types are parametric in their option type, this option hierarchy can directly mirror any subset relationship Tk defines, without causing any unwanted effects on the widget hierarchy. Moreover, the subtype axiom `WWidget a < Packable` effectively ensures that any pair of packable widgets can be combined by the layout combinators, irrespective of their defined options. A sample of the different widget options available in Tk follows below.

```
data WidthOpt =
      Width Int
```

```
data CanvasOpt > WidthOpt =
       Background Color
     | Borderwidth Int
     | Height Int

data WindowOpt > CanvasOpt =
       Title String

data LabelOpt > CanvasOpt =
       Foreground Color
     | Font String
     | Text String
     | Anchor AnchorType

data ButtonOpt > LabelOpt =
       Command Action
     | State StateType

data LineOpt > WidthOpt =
       Fill Color
```

Datatype subtyping is also put to good use in the definition of mouse events, as shown below.

```
data ButtonPress =
       ButtonPress Int (Pos -> Action)
     | AnyButtonPress (Int -> Pos -> Action)

data MouseEvent > ButtonPress =
       ButtonRelease Int (Pos -> Action)
     | AnyButtonRelease (Int -> Pos -> Action)
     | Motion (Pos -> Action)
     | Double ButtonPress
     | Triple ButtonPress

data Event > MouseEvent, KeyEvent, WindowEvent
```

The root type of the Tk interface, finally, is a record type defined as follows:

```
struct TkEnv =
   window    :: [WindowOpt] -> Request Window
   delay     :: Int -> Action -> Action
   periodic  :: Int -> Action -> Action
   dialog    :: String -> [(String,Action)] -> Action
   bell      :: Action
   quit      :: Action
   ...
```

Here `delay`, `periodic`, and `dialog` are true reactive variants of some of the few predefined Tk scripts that bypass the event-driven principle and involve indefinite blocking. Our signatures should particularly be noted for the callbacks they take as arguments. A nice consequence of our reactive dialog-box implementation is that all objects are alive as usual while the box is open, thus the responsibility of screening out irrelevant mouse events is put where it belongs, i.e. on the window manager.

Since the `main` procedure can be assumed to take a `TkEnv` as an argument, the whole O'Haskell interface to Tk can actually be described in terms of a set of type declarations, and a set of purely functional layout combinators. We believe that this clarity and explicit structure represents an immediate advantage over other strongly typed Tk interfaces, notably the one presented by TkGofer [CMV97].

3.2 An AGV controller

Our next example clearly illustrates the separation between the calculations performed by a program, and interactions in which it is involved. Since it is an implementation of an interrupt-driven system with parallel processes which also performs heavy calculations, it captures many of the characteristics of a typical embedded system.

The concrete task is to control an autonomously guided vehicle (AGV). Such a vehicle is capable of navigating (within a limited area) without the guidance of a human driver. The crucial point for an AGV is to know its position at all times. There are different methods developed for this purpose, the one we will consider relies on the existence of a set of *reflectors* placed out at known positions within the room in question [Hyy87]. To determine its position, the AGV measures angles to the visible reflectors by means of a rotating laser beam. The angles are then compared with the positions of the reflectors, to yield a position within the room from which the angles must have been measured. If this position does not coincide with the desired position at that time, a regulating algorithm generates appropriate adjustments to the driving and steering servo.

A first step in programming a controller for such an AGV is to separate the two major algorithms — positioning and regulation — from the questions of environment interaction. The algorithms, called `calcpos` and `regulate`, are naturally coded as functions, but since their implementation mainly concerns matters that belong to the field of control theory, we only give their type signatures.

Viewed from the outside, the AGV system naturally decomposes into four main objects: the scanner, the simulated driver, the radio receiver that interprets mission commands, and the servo. Of these objects, we only give code for the first two; the others should be straightforward to implement given adequate hardware and protocol details.

The code for our AGV example follows below.

```
   type Angle    = Float
   type Velocity = (Angle,Float)
   type Pos      = (Float,Float)

   calcpos  :: [Angle] -> [Pos] -> Pos
   regulate :: Pos -> Pos -> Velocity -> Velocity
   room     :: [Pos]

   struct Driver =
      newscan :: [Angle] -> Action
      newpath :: [Pos] -> Action

   driver servo =
      template
         velocity := (0.0,0.0)
         path     := repeat (0.0,0.0)
      in struct
         newscan angles = action
            let pos:rest = path
               pos1     = calcpos angles room
            velocity := regulate pos1 pos velocity
            path  := rest
            servo.set_velocity velocity
         newpath p = action
            path := p

-----------------------------------------------------
   struct Scanner =
      detect     :: Action
      zerocross :: Action

   scanner port driver =
      template
         angles := []
      in struct
         detect = action
            a <- port.load
            angles := 2*pi*(fromIntegral a)/4000 : angles
         zerocross = action
            driver.newscan angles
            angles := []

-----------------------------------------------------
   struct Servo =
      set_velocity :: Velocity -> Action
```

```
   servo :: Port -> Port -> Template Servo

   struct Radio =
      incoming :: Action

   radio :: Port -> Driver -> Template Radio

-------------------------------------------------
   struct Port =
      load  :: Request Int
      store :: Int -> Action

   struct SysEnv =
      reserve_port   :: Int -> Request Port
      install_vector :: Int -> Action -> Action

   main env = do
      steer_port <- env.reserve_port 0xFFFF0001
      velo_port  <- env.reserve_port 0xFFFF0002
      angle_port <- env.reserve_port 0xFFFF0003
      comm_port  <- env.reserve_port 0xFFFF0004

      serv <- servo steer_port velo_port
      driv <- driver serv
      scan <- scanner angle_port driv
      comm <- radio comm_port driv

      env.install_vector 0x80 scan.detect
      env.install_vector 0x81 scan.zerocross
      env.install_vector 0x82 comm.incoming
```

The AGV controller must obviously be a sampling system, and the clock driving this implementation is actually the rotating beam, which issues an interrupt (tick) at the completion of each turn (as well as another interrupt at the detection of a reflector). At each tick, the scanner process sends the detections of the completed scan to the driver by the triggering of action `newscan`, whose implementation is greatly simplified by the ability to encode complex calculations in the functional domain.

The state of the driver process consists of the current velocity vector, and a list of positions that constitute the path that the AGV has to follow. The path list is consumed one element per tick, and in order to make our example simple, we have assumed that the list is infinite. Thus, to make the AGV stop, the path must contain the same position repeatedly. Note that since action activations for an object are automatically serialized, the radio receiver can safely trigger `newpath` at any time to give the AGV a new path to follow.

In the program body `main` we touch on the issue of how concrete hardware

interaction in an embedded system can be expressed. Assuming that the specific machine instructions required to access ports and the interrupt-vector can be hidden behind the bare-bones computing environment interface `SysEnv` defined above, the `main` routine (i.e. the reset-handler) of the embedded system proceeds not unlike the initiation of an ordinary interactive program, by creating resources and installing event callbacks (interrupt-handlers in this case). Concrete issues like word sizes and data representations are however avoided in this example, for the sake of brevity.

3.3 Sieve of Eratosthenes

The "Sieve of Eratosthenes" is a classical method for computing prime numbers. Its underlying idea is that a list of all prime numbers less than n can be obtained from a list of all numbers in the range $[2..n]$, by removing all multiples of the first number, then removing all multiples of the second number in the resulting list, and then the multiples of the third number, etc, until all numbers in the original list have had their multiples removed. The resulting list must then consist only of numbers that cannot be factorized, i.e. the list contains only the prime numbers (besides 2).

A recursive function that implements this algorithm is easy to define. The challenge is instead to exploit the parallelism inherent in the algorithm, so that all screening calculations can run on a dedicated processor if hardware so permits.

Like many other advocates of purely functional languages, we take the standpoint that performance-enhancing parallelism preferably should be *semantically transparent*; that is, only visible in the code in the form of carefully placed annotations containing semantically irrelevant scheduling hints [Ham94]. However, as an evidence that the concurrency features available in O'Haskell actually are up to the task of expressing explicitly parallel algorithms as well, we will show an implementation of Sieve of Eratosthenes in which every found prime number is represented by a concurrent object.

The code for our implementation looks as follows.

```
struct Cell =
   feed :: Int -> Action

cell print n =
   template
      next  := Nothing
   in struct
      feed k = action
         if k `mod` n /= 0 then
            case next of
               Nothing ->
                  c <- cell print k
                  next := Just c
```

```
                  print (show k ++ " ")
               Just c ->
                  c.feed k

sieve print n = do
   c <- cell print 2
   forall i <- [3..n] do
      c.feed i

main env = sieve env.putStr 10000
```

Two things are worth noticing. Firstly, the cell objects created by this program form a chain of filter processes at all times, terminated by a cell whose internal state equals `Nothing`. Secondly, each cell runs its asynchronous method `feed` in parallel, yet the output written on the screen will be deterministic thanks to the order-preserving semantics of method invocation in O'Haskell.

3.4 Semaphore encoding

A semaphore is a synchronization device that in its simplest form is just a "token" that can be claimed and released, and where the claiming operation may block if the token is not available [Dij65]. According to our definition, a semaphore is clearly not a reactive abstraction.

However, there might be situations where the implementation of synchronization mechanisms is actually a part of the programming problem, for example in the simulation of a railway system. Hence it might be necessary to encode semaphores reactively, and we need to show how this can be done.

The key step towards a reactive implementation is to lift out the client code that is supposed to run after a successful claim, and put it into a separate action. Then the responsibility for triggering this action can be put on the semaphore, quite similar to the *continuation passing style* sometimes used in functional languages to encode stateful computations.

Here is the semaphore implementation:

```
struct Semaphore =
   claim   :: Action -> Action
   release :: Action

semaphore :: Template Semaphore
semaphore =
   template
      active := False
      wakeup := []
   in struct
      claim grant  =  action
         if not active then
```

```
            active := True
            grant
         else
            wakeup := wakeup ++ [grant]

      release = action
         case wakeup of
            []   -> active := False
            w:ws -> wakeup := ws
                    w
```

A client of such a semaphore typically invokes the semaphore operations as follows:

```
do ...
   sem.claim (action
                 cmds
                 sem.release)
```

Syntactically this style is actually not very much different from the traditional pattern of semaphore use that relies on a blocking claim operation. Consider for example the same fragment written in a traditional style:

```
do ...
   sem.claim
   cmds
   sem.release
```

The difference between this code and our reactive formulation is of course that the reactive example allows the implied client object to service other messages while it waits for the semaphore claim to be granted. Achieving something similar in the traditional case would evidently require some form of object-internal concurrency in order to be expressible.

One can imagine many variations on the encoding above, e.g. letting `claim` be a Boolean **request** that returns `True` in the successful case instead of triggering `grant`. This would allow a client to take special action depending on whether the claim can be granted immediately (just like a real train driver does when a semaphore indicates that the train can pass without first reducing its speed to zero).

3.5 Queue encoding

An alternative to the many-to-one buffering mechanism inherent in O'Haskell would be a many-to-many communication scheme, where data is communicated via a distinguished *queue* process. In order to implement such a queue in O'Haskell, however, we need to reconsider the *remove* operation, since it is supposed to block if no data is available. The reactive way of removing an

item must be a two-phase operation: first a consumer *announces* its readiness to receive some data, then, perhaps later on when data has arrived, the queue process sends a message with the data to the consumer process. The code for a reactive queue looks as follows:

```
struct Queue a =
   insert   :: a -> Action
   announce :: (a -> Action) -> Action

queue :: Template (Queue a)
queue =
   template
      packets := []
      servers := []
   in struct
      insert p = action
         case servers of
            []   -> packets := packets ++ [p]
            s:ss -> servers := ss
                    s p

      announce s = action
         case packets of
            []   -> servers := servers ++ [s]
            p:ps -> packets := ps
                    s p
```

Note that this "polarity switch" of the remove operation does not really clutter up the code; it is still as symmetrical as it would be in a language with blocking read operations. Turning the queue into a bounded buffer would mainly just require switching the polarity of the insert operation as well. The real benefit of this reactive encoding is that all processes involved in communication with a queue can easily be extended to handle simultaneous interactions in other directions as well, without this having any effect on the basic queue communication pattern.

3.6 Delay timer

As an example of how the concept real time and delays fits into the reactive style, we give an implementation of a timer process, that allows its clients to "sleep", or delay its activities for a specified number of ticks. Sleeping is here interpreted reactively, however, meaning that a process passively awaits *any* message, one of which will signal that a certain amount of time has passed. This particular action must be supplied as a parameter to the timer each time the delay service is demanded. Even though we present the interface to the timer process as a record value, it is implied that the `tick` action should be

installed in some interrupt vector table, and `start` should preferably be made available through some general, operating-system-like interface, like for example the `TkEnv` interface described in section 3.1.

```
struct Timer =
   start :: Int -> Cmd () -> Action
   tick  :: Action

timer  =
   template
      time := 0
      pend := []
   in let
      start t cmd = action
         pend := insert (time+t,cmd) pend

      check_pending = do
         case pend of
            (t,cmd):pend' | time >= t ->
               pend := pend'
               cmd
               check_pending
            _ ->
               done

      tick = action
         time := time + 1
         check_pending

   in struct ..Timer
```

A notable feature of this example is the use of a recursive procedure in the interrupt service routine `tick`. Recall that such a call is completely local to a process, and cannot be interspersed with other messages.

It is interesting to see that safe communication between an asynchronous interrupt service routine and an ordinary process (a rather tricky task in most programming languages) can be handled with the same mutual exclusion machinery that is used in ordinary interprocess communication. The reason behind this is that, in effect, *all* messages are modeled as interrupts in O'Haskell (interrupting the normal, passive state of an object), and it does not matter whether some of them are actually generated by hardware.

3.7 A telnet client

We will now continue with a slightly larger example: a rudimentary implementation of a Telnet client. The resulting code has a very appealing structure, cen-

tered around the intuitive fact that the state of a Telnet process is its current Tcp connection. We have not implemented any Telnet-specific handshaking, though, since that would just mean repeating the pattern used in the handling of open/close acknowledgements.

```
struct Telnet < StdProg =
   connect    :: Host -> Action
   disconnect :: Action

struct Client =
   connected :: Connection -> Action
   deliver   :: Packet -> Action
   closed    :: Action

telnet :: Tcp -> StdEnv -> Template Telnet
telnet tcp env =
   template
      you_server := Nothing
   in let me_client = struct
      connected c = action
         you_server := Just  c
         env.putStr "[Connected]\n"

      deliver pkt = action
         env.putChar (mkchar pkt)

      closed = action
         you_server := Nothing
         env.putStr "[Disconnected]\n"

   in struct
      connect host = action
         tcp.open host telnet_port me_client

      getChar ch = action
         case you_server of
            Just c  -> c.send (mkpkt ch)
            Nothing -> env.beep

      disconnect = action
         case you_server of
            Just c  -> c.close
            Nothing -> done

      signal n = action
         done
```

```
struct Tcp =
   open   :: Host -> Port -> Client -> Action
   listen :: Port -> Client -> Action

struct Connection =
   send  :: Packet -> Action
   close :: Action
   peer  :: Request Host
```

Notice how the Telnet process exhibits two different interfaces at the same time, depending on which level in the protocol hierarchy it is seen from. The type definitions used in this example also demonstrate a principle we would like to call *communication by contract*, whereby the commitments of both peers involved in a dialogue are clearly stated in terms of the record types that the peers either expose or assume. Particularly note how this principle is manifested in the fact that the use of a Tcp connection is effectively prohibited until it is established (by the splitting of the interface to Tcp into two record types, `Tcp` and `Connection`).

Finally, one interesting consequence of the reactive implementation is actually visible in the body of `getChar`, where the user is notified by a beep if characters are entered before a reliable connection has been established. This would not have been so straightforward in a language that had modeled `tcp.open` as a blocking input operation.

3.8 A telecom application

Our last example is an O'Haskell adaption of a very illustrative program fragment taken from the *Erlang* book [AVWW96]. The code in question implements the central part of a telephone exchange; more specifically it provides a generic template for all the processes in such an exchange that are assigned the task of controlling an individual phone line. In the telecom jargon such a functionality goes under the name Plain Ordinary Telephony Service, or *POTS* for short.

A POTS process operates in the environment provided by a *tele-os*, which is an abstraction of the primitive services a phone line controller might need. Included in this set of services are the commands that control ringing and tone signalling, the methods that set up connections within the actual switching hardware, and a method which looks up the interface of another POTS process on basis of a supplied telephone number. It is also assumed that there is a delay timer available for the purpose of generating timeouts. This timer can essentially be of the kind described in section 3.6, with the addition that it also returns the current time each time it is set up, as a tag for identifying subsequent timeouts. It is assumed here that the precision of the clock is sufficient to guarantee that all such tags are unique; should that not be the case it is straightforward to supplement the `Time` datatype with an extra tag component that achieves the same effect.

The types describing the tele-os interface follow below.

```
struct TeleOS =
   start_ring  :: Address -> Action
   stop_ring   :: Address -> Action
   connect     :: Address -> Address -> Action
   disconnect  :: Address -> Address -> Action
   start_tone  :: Address -> ToneType -> Action
   stop_tone   :: Address -> Action
   analyse     :: [Digit] -> Request NumResponse
   start_timer :: Time -> (Time -> Action) -> Request Time

data ToneType = Dial
              | Ring
              | Busy
              | Fault
              | None

data NumResponse = Invalid
                 | MoreDigits
                 | Valid PotsB Address
```

A POTS process, in turn, presents three different interfaces depending on who the client of its services is. Seen from the telephone hardware, a POTS process looks like this:

```
struct PotsHW =
   off_hook   :: Action
   on_hook    :: Action
   digit      :: Digit -> Action
```

When two POTS processes talk to each other, the following two interfaces are used instead:

```
struct Pots =
   cleared    :: Action

struct PotsA < Pots =
   answered   :: Action

struct PotsB < Pots =
   seize      :: PotsA -> Request Bool
```

Here the A and B suffixes are used to distinguish the interfaces exposed by active and passive POTS processes, respectively. Every POTS process can take on both these roles, since any normal telephone can be used for both outgoing and incoming calls. Notice that the interfaces stored in the tele-os phone number database are all of the passive type `PotsB`.

A POTS process can be described as a state machine that alters between six major states. This is captured in a datatype in our implementation.

```
data PotsState = Idle
               | GetNumber   [Digit]
               | Ringing     PotsB Address Time
               | Incoming    PotsA
               | Speech      Pots (Maybe Address)
               | WaitOnHook  ToneType
```

We are now ready to give the code for the POTS template. The internal state of a POTS process is a value of type `PotsState`, and the implementation task is essentially to define an appropriate reaction for each method/state-value combination. We show the code in one piece, and add some comments afterwards.

```
pots myaddr tele_os =
   template
      state  := Idle
   in let

      off_hook = action
         case state of
            Idle ->
               tele_os.start_tone myaddr Dial
               state := GetNumber []
            Incoming p ->
               tele_os.stop_ring myaddr
               p.answered
               state := Speech p Nothing
            _ -> done

      on_hook = action
         case state of
            GetNumber [] ->
               tele_os.stop_tone myaddr
            Ringing p adr time ->
               p.cleared
               tele_os.stop_tone myaddr
            Speech p Nothing ->
               p.cleared
            Speech p (Just addr) ->
               p.cleared
               tele_os.disconnect myaddr addr
            WaitOnHook tone_type | tone_type /= None ->
               tele_os.stop_tone myaddr
            _ -> done
```

```
   state := Idle

digit d = action
   case state of
      GetNumber nr ->
         if null nr then
            tele_os.stop_tone myaddr
         let nr' = nr ++ [d]
         resp <- tele_os.analyse nr'
         case resp of
            Invalid ->
               tele_os.start_tone myaddr Fault
               state := WaitOnHook Fault
            MoreDigits ->
               state := GetNumber nr'
            Valid p adr ->
               ready <- do p.seize (struct ..PotsA)
                         handle
                            Deadlock -> return False
               if ready then
                  tele_os.start_tone myaddr Ring
                  tag <- tele_os.start_timer (Sec 90)
                                             timeout
                  state := Ringing p adr tag
               else
                  tele_os.start_tone myaddr Busy
                  state := WaitOnHook Busy
      _ -> done

answered = action
   case state of
      Ringing p adr _ ->
         tele_os.stop_tone myaddr
         tele_os.connect myaddr adr
         state := Speech p (Just adr)
      _ -> done

timeout tag = action
   case state of
      Ringing p adr tag' | tag == tag' ->
         p.cleared
         tele_os.stop_tone myaddr
         tele_os.start_tone myaddr Fault
         state := WaitOnHook Fault
      _ -> done
```

```
   cleared = action
      case state of
         Incoming p ->
            tele_os.stop_ring myaddr
            state := Idle
         Speech p Nothing ->
            state := WaitOnHook None
         Speech p (Just adr) ->
            tele_os.disconnect myaddr adr
            state := WaitOnHook None
         _ -> done

   seize p = request
      case state of
         Idle ->
            tele_os.start_ring myaddr
            state := Incoming p
            return True
         _ ->
            return False

in (struct ..PotsHW, struct ..PotsB)
```

Informally this code can be understood as follows.

- When an `off_hook` signal is received, the POTS process enters a dialing phase if it was idle, or it enters the speech state if it was signaling an incoming call.
- When an `on_hook` signal is received, the POTS process resets itself in slightly different ways depending on its current state.
- When a digit is received during a dialing phase, the tele-os is requested to analyse the accumulated digit sequence. Depending on the outcome of this analysis, the POTS process either
  - signals an error and enters a cleanup state (`WaitOnHook`) if the number is invalid.
  - continues the dialing phase if more digits are expected.
  - tries to contact a peer process if the number is complete. This attempt either fails, in which case a busy tone is signaled, or it succeeds, in which case a timer is started and ringing tone is presented to the user.
- If an answer is detected when a ringing tone is present, a connection is opened in the switching hardware and the speech state is entered.

- If a timeout occurs in the ringing state, an error is signaled and the cleanup state is entered.
- If the other party sends a `cleared` message, the POTS process goes idle if a speech connection has not yet been established, otherwise it first passes the cleanup state.
- An incoming call is accepted only if the POTS process is idle, otherwise the call is rejected.

This scenario only describes the normal chains of events. However, in reality *any* combination of events/states must be expected, since it is absolutely vital for the operation of a telephone exchange that a POTS process cannot be fooled into a bad state if (say) the user for some reason suddenly starts entering digits when a connection is already open. Likewise, if all hardware is functioning properly it should not be possible to receive an `on_hook` signal in the `Incoming` state, since such a signal must reasonably have been preceded by an `off_hook` which terminates the `Incoming` state. However, telephones occasionally break down, and it cannot be tolerated that the correct behaviour of a telephone exchange depends on the absence of such trivial failures.

For these reasons, POTS processes *must* be reactive. This is of course an obvious truth in the telecom industry, and the original Erlang code that this example is based on is also carefully constructed to uphold the reactive property. In Erlang this requires active cooperation from the programmer, though, since default clauses must be added to every receive operation to ensure that messages not specifically handled are nevertheless consumed (but ignored). O'Haskell, in contrast, upholds the reactive property *by default*, by not allowing the programmer to forget (or disable) reception of any kind of message in any state. On the other hand, O'Haskell programmers must make sure that pattern-matching against the local state is exhaustive, but any failure to do this is statically detectable, and should be reported as a warning by every decent language implementation.

Interestingly, the original Erlang code actually deviates slightly from the commitment to serve any kind of message at one point. This concerns the case when a `seize` message has been sent to a peer process, and a positive or negative reply is to be collected. Here the Erlang implementation chooses to accept only the expected replies, plus a concurrently incoming `seize` message (which is the Erlang way of detecting and resolving a deadlock). This kind of coding can only be understood if one assumes that the peer process is *local* in the sense that it cannot fail without causing a local system failure that includes the original POTS process as well. That is to say that the POTS processes are assumed not to be subject to partial failures in this example (in sharp contrast to all the unpredictable telephone users on the network!). Under this assumption, bi-directional communication is more adequately expressed as a synchronous request in O'Haskell, and as can be seen in the code, the deadlock case can then be conveniently caught by the general exception-handling mechanism of the language.

Alternative formulation

One evident difference between our implementation of the POTS example and the Erlang original given in [AVWW96] is that the pairing of events with states is organized along two different axes. In our encoding we start by enumerating all possible events (methods), and then for each event we describe the actions taken for each possible local state. The Erlang code is organized the orthogonal way: the states are enumerated as a set of mutually recursive functions, and in each function the appropriate reaction to every event is defined.

Both ways of organizing reactive code have their merits, even though we believe that the O'Haskell way has a more definite object-oriented "flavour". However, as an illustration to the possibility of actually choosing the most suitable organization principle when programming in O'Haskell, we give an alternative encoding of the POTS example in appendix B. This code has even more similarities with the Erlang master, although it is still based around a template expression of the same type as before. Some notable highlights in this encoding are:

1. The use of a recursive record type to express a local state that contains procedures depending of the same state.

2. The extensive use of record stuffing to achieve convenient access to default procedure implementations.

We conclude that apart from being statically typed and reactive by design, O'Haskell provides a degree of coding freedom that Erlang lacks. Still, it is interesting to see that Erlang code — at least for what the current example is concerned — can be straightforwardly translated into O'Haskell irrespective of which organization principles that is chosen.

Chapter 4

Case study: A modular implementation of self-stabilization

Self-stabilization is a quality attributed to network protocols that are guaranteed to exhibit correct behavior in a finite amount of time, even if started in an erroneous state. Such protocols clearly have some very desirable fault-tolerant properties, e.g. the ability to automatically recover from arbitrary state-variable corruption, in any number of nodes, once memory failures stop. Self-stabilizing protocols also avoid some of the initialization problems that are inherent in distributed and dynamic systems, since, in effect, any network state is an acceptable initial state.

The concept of self-stabilization was originally conceived by Dijkstra [Dij74]. While the ensuing work has mostly been concerned with the construction of self-stabilizing protocols for particular tasks like token passing, network reset, and spanning tree construction [BP89, GM91, KP93, DIM93, IJ90, AG94, AKY90], some general methods for transforming existing protocols into self-stabilizing counterparts have also been devised. The scheme proposed by Katz and Perry can convert an arbitrary non-stabilizing protocol into a stabilizing one; however, this generality comes at the expense of a quite substantial space and processing overhead that must be paid even during normal operation [KP93]. In contrast, Awerbuch, Patt, and Varghese are able to construct highly efficient stabilizing protocols by their transformations, but their method is limited to protocols that are amenable to what is called *local checking and correction* [APSV91, Var93]. Unfortunately, both these approaches are described using a mixture of formal and informal arguments, and it is not entirely clear from the sources available how to put the ideas into practice, given an implementation of some particular non-stabilizing algorithm.

In this chapter we will leverage on O'Haskell in order to obtain a more concrete implementation of the self-stabilizing method proposed by Awerbuch,

Patt, and Varghese. Their method has a quite natural encoding in O'Haskell, and we present it primarily as a case study of the usefulness of O'Haskell as a network programming language. Besides that, however, we also think that our coding exercise makes some small contributions to the field of self-stabilization:

- It sheds some light on coding issues that are either just outlined in [APSV91], or obscured by the rather abstract formalism of I/O Automata used in [Var93].

- It provides *modularity*, in the sense that the self-stabilizing code is encapsulated and can be plugged into the signal path of any protocol implementation that meets the requirements of local checking and correction. This stands in contrast to the original method description, which is specified as stepwise *modifications* that must be made to the non-stabilizing protocol code.

Despite these contributions, though, we will refrain from making any claims that our code can actually be used for its purpose as it stands. This somewhat unusual caveat is necessary because self-stabilization in the context of a higher-order and non-strict language seems to be a rather new combination, whose implications are far from understood. In particular, the mix-up of code and data that results from common heap-based implementation techniques seems to directly invalidate the main assumption that underlies the self-stabilizing model: only program state can be corrupted, not program code. We are not saying that these problems cannot be addressed using alternative techniques, but in the absence of a thorough analysis of the self-stabilization hypothesis in the context of functional languages, we must consider our case study more of an illustration of the expressive power of O'Haskell, than a serious attempt to produce a practically useful algorithm. If the implementation still needs some practical motivation, it is probably sufficient to restrict the assumed transient errors to just spontaneous *node resets* in order to obtain a practically valid self-stabilizing algorithm as well.

The chapter now continues with an introduction to the ideas behind local checking and correction. Section 4.2 lays down some basic interface definitions for programming network protocols in O'Haskell, before we give a swift overview of I/O Automata and their relation to our language (section 4.3). Section 4.4 describes a template `stabilizer`, which is an O'Haskell encoding of the general self-stabilizing mechanisms that implement local checking and correction. In section 4.5 we provide an implementation of a protocol example from [APSV91], and show how it is made self-stabilizing by means of our stabilizer in section 4.6. The topic of inheritance is revisited in the light of this case study in section 4.7, before the chapter rounds up with our conclusions in section 4.8.

4.1 Self-stabilization by local checking and correction

In the literature on local checking and correction, a network is a directed, symmetric graph, where each node is modeled as a *node automaton* (in the I/O Automata formalism), and each link is a *unit storage data link* (UDL), that is also modeled as a simple I/O automaton. A *link subsystem* is defined as the composition of two connected node automata, together with the two unidirectional UDLs that connect them. Thus, a link subsystem is uniquely identified by a pair of node identifiers.

The key idea behind local checking is now this: for many protocols it can be shown that whenever the protocol is in an illegal state, some link subsystem must also be in an illegal state. So instead of collecting and checking state information for the whole network at some central node, a protocol amenable to local checking can examine each link subsystem in parallel. A corresponding conception underlies the theory of local correction: many protocols can return to a legal state by having each link subsystem independently correct itself to a legal state. An intuitive precondition here is that correcting one link subsystem must not invalidate the correctness of others, however, it can be shown that a sufficient condition is to require that any such invalidation dependencies be acyclic.

The goal of the transformation scheme of Awerbuch, Patt, and Varghese is to convert any locally checkable and correctable protocol into a self-stabilizing variant. To this end it is assumed that for the given, non-stabilizing protocol, there is a set of local predicates $L_{u,v} = L_{v,u}$ that can be used for checking every link subsystem (u, v), and a correction function $f_{u,v}$ for resetting the state of a node automaton u w.r.t. the link subsystem (u, v).

To simplify the model it is assumed that in each link subsystem there is a designated *leader* node that is responsible for periodically checking, and eventually correcting the link subsystem. As a further simplification it is prescribed that the UDLs of a link subsystem be empty during checking and correction, hence ordinary traffic must be suspended whenever the subsystem is in a checking/correction phase. In [Var93] this is achieved by means of Boolean flags in the node automata, under the supervision of code added by the transformation scheme. We will discuss an alternative solution to this problem in the next section.

The essence of the checking and correction extension is quite simple. For a particular link subsystem (u, v), the leader node (say u) sends out a distinguished *snapshot request* packet to its neighbour node (v), which in turn replies with a *snapshot response* packet containing its current state. Node u then uses local predicate $L_{u,v}$ to determine whether correction of (u, v) is needed. If so, a *reset request* packet is sent out by u, which causes v to reset itself using $f_{v,u}$ and send back a *reset response* packet. When node u receives the response, it also resets its state using $f_{u,v}$.

The correctness of this snapshot/reset algorithm relies on the assumption

that the given protocol is sound in the sense that legal states are only followed by legal states (in the absence of memory faults). Hence $L_{u,v}$ and $f_{u,v}$ can be used as if they were applied to the whole link subsystem (u, v) at once, even though the individual node states are sampled at different times. In order to make the snapshot/reset algorithm insensitive to garbage packets on the links (i.e. making the stabilizing code itself self-stabilizing), *counter-flushing* is used (see [Var93]). Since the communication channels are UDLs, there can only be 3 distinct packets hidden in a link subsystem; hence counters can wrap around when they reach 4.

Further details on the theory of local checking and correction, as well as proofs of the formal claims made, can be found in [Var93]. We will now proceed with the encoding of these ideas in O'Haskell.

4.2 Some basic definitions

As an introduction to our O'Haskell encoding of self-stabilization we define a small hierarchy of record types that will form the basis of the object interfaces to follow.

```
type Link = Int

struct Client a =
   receive :: Link -> a -> Action

struct PolledClient a < Client a =
   poll :: Link -> Request (Maybe a)

struct AckClient a < Client a =
   ack :: Link -> Action
```

These definitions are intended to capture the operations exported by various kinds of communication protocols, as seen from an immediate *lower* layer in the protocol stack. For the particular application of this chapter, the addressing needs are rather simple, thus we just parameterize each operation with respect to the index number of a particular link.

It turns out that most protocols intuitively expose a quite different interface to their upper clients, from what is visible from below. We can express this by means of another, unrelated set of record definitions:

```
struct NullNetwork

struct Network a < NullNetwork =
   send :: Link -> a -> Action
```

Here we see an example of a record type without any selectors. This definition is useful since there are protocols (e.g. those that wish to poll their client for

outgoing data) that do not export any operations at all for the upper layers to invoke.

As we have seen, the interface to an O'Haskell object can be any type, in particular it can be a pair of two distinct sub-interfaces intended for two different kinds of users. We establish this as a convention for network programming in O'Haskell, and define a protocol type `Layer`, and a general layer combinator `<||>`, for stacking two protocols and connecting their respective interfaces:

```
type Layer hi ho li lo = (ho,lo) -> Template (hi,li)

proto_a <||> proto_b = \(ho,lo) =
   do fix (hi,mi) <- proto_a (ho,mo)
          (mo,li) <- proto_b (mi,lo)
      return (hi,li)
```

Recall from section 2.6 that the **fix** keyword denotes a syntactic extension to Haskell that hides the use of a fixpoint operator on the monad level. Such an operator becomes necessary the moment we want to instantiate objects with mutually recursive dependencies, which is clearly the case here: protocol a depends on the upper interface to protocol b, which in turn needs access to the lower interface to protocol a. The type of the stacking combinator becomes

```
(<||>) :: Layer hi ho mi mo
       -> Layer mo mi li lo
       -> Layer hi ho li lo
```

4.3 O'Haskell vs. I/O Automata

Before we proceed to the actual encoding of self-stabilization, we need to make a few comments on the relationship between O'Haskell and the formalism used in the original description of the algorithm: Input/Output Automata, or IOA for short [LT89].

First, we notice that there are indeed striking similarities between the formalisms: objects in O'Haskell resemble automata both in terms of state encapsulation and atomicity of operations, and the constantly enabled input actions of IOA have a direct counterpart in O'Haskell. For this reason, it will be clear to readers familiar with the literature on self-stabilization, that our implementation resembles the original notation in many ways.

However, closer scrutiny reveals a crucial difference between the two models in the way actions are triggered. In the case of IOA, an input action is executed whenever the preconditions of a corresponding *output* action makes that action enabled. Normally, output actions are defined to become disabled as an effect of their execution, but nothing in the IOA model really prevents an output action from being *continuously* enabled. Such a state will actually result in an arbitrary number of action activations, as long as the precondition holds for the output action in question.

In O'Haskell there are no output actions, the closest equivalent must instead be the hypothetical actions we can introduce by lambda abstraction (c.f. the parameters `ho` and `lo` in the definition of `<||>`. These actions, as well as any action for that matter, are only invoked explicitly in response to some specific event. In particular, an action cannot be activated by the static condition that some predicate *is* true; it must instead be triggered by the state-changing event that makes the predicate *become* true.[1]

This semantic difference has a direct consequence to our encoding of the UDL interface. In [Var93], a UDL is modeled as an automaton with an input action SEND and an output action FREE, which is enabled whenever the UDL is ready to accept new packets (the definition of the UDL automaton is provided in appendix C, figure C.1). In order to know when a packet can be sent, a client of the UDL is required to input the FREE action as an acknowledgment, so it can keep an internal flag that mirrors the state of the UDL. This scheme works even if the flag is erroneously set to false, since FREE is continuously enabled by the UDL.

However, a direct encoding of this UDL model in O'Haskell (e.g. as a template of type `(AckClient a, b) -> Template (Network a, c)`) would not be self-stabilizing, because a memory fault in the client that resets the free flag just after an acknowledgment has been received, irrevocably leads to a passive deadlock. Moreover, mimicking the IOA semantics by having the UDL repeatedly send acknowledgments (invoking `ack`) when it is ready, implies race conditions that may result in loss of data. An alternative would be to turn both `send` and `ack` into synchronous requests, but then active deadlock can occur, and its proper handling would unduly obscure the client code.

Instead, in a self-stabilizing context it is arguably better to simply make the UDL repeatedly ask for packets when it is ready, i.e. to use *polling*. The frequency of these requests can be controlled by a timer to enable fine-tuning of the load that the polling mechanism should incur on the system. Indeed, such a polling implementation should have essentially the same dynamic behavior as an imagined implementation of an IOA, where a scheduler would need to assert that a continuously enabled action is repeatedly executed at some suitable interval.

A detailed, self-stabilizing implementation of the UDL layer itself is beyond the scope of this case study, thus we will only consider its interface henceforth (see e.g. [APSV91] for some implementation details). In doing so we will in fact abstract a bit further, and assume that the interface to the UDL layer is bidirectional, and that it also takes care of routing packets to and from the actual objects controlling the individual links. This makes it possible to treat the UDL layer as a stackable protocol, with the following type:

```
udl :: Int ->
       (PolledClient a, HWNetwork) ->
       Template (NullNetwork, HWClient)
```

[1] For an analogy, we can think of an IOA as a *gated* digital circuit, while an O'Haskell system bears more resemblance to *edge-triggered* circuit technology.

Here we assume that the integer argument determines the number of links managed by the UDL layer, and that `HWNetwork` and `HWClient` are some abstract types suitable for interfacing with the communication hardware.

4.4 Implementing self-stabilization

We are now ready to discuss the actual self-stabilizing code. In our implementation, it is built up as a separate protocol layer, that can be inserted between any stabilizable protocol and the bottom UDL layer. Since messages to and from the network this way are intercepted by the stabilizer, traffic can effectively be subsumed when the stabilizer enters a checking/correction phase.

To simplify matters, we assume that the given stabilizable protocol is prepared to be polled for outgoing data, as it would have to be in a non-stabilizing setting when stacked on top of the UDL abstraction. This restriction is not very serious, since it is straightforward to implement buffering in an upper layer should an asynchronous sending interface be desired (we will actually see an example of this in section 4.5). However, an imperative demand on the protocol in question is that it is locally checkable and correctable, in the sense discussed in section 4.1. This is captured in O'Haskell by requiring the client of our stabilizer to export the following interface:

```
struct StabilizableClient a s < PolledClient a =
   current_state    :: Link -> Request s
   check_invariants :: Link -> s -> Request Bool
   local_reset      :: Link -> Action
```

The local predicate $L_{u,v}$ and the local correction function $f_{u,v}$ are here called `check_invariants` and `local_reset`, respectively.

Injection of *request* and *response* packets into the data stream is achieved by means of a datatype:

```
data StablePkt a s = DataPkt     a
                   | RequestPkt  Int SnapMode
                   | ResponsePkt Int SnapMode s

data SnapMode = Snap | Reset
```

The `Int` parameters in these packets are the counter values that implement counter flushing.

To enable local checking and possibly correction at regular intervals, we will define the lower interface to the stabilizer layer to be of type `PeriodicClient` (which in turn is defined using a type `Periodic`, since the periodic property might be desirable to add to other types in the interface hierarchy as well).

```
struct Periodic =
   tick :: Link -> Action

struct PeriodicClient a < PolledClient a, Periodic
```

It is implied here that the constructor of a particular protocol stack will connect the `tick` method of our stabilizer to a properly initialized array of timers, one for each link.

The actual code for the stabilizer looks as follows:

```
stabilizer :: Int ->
              Array Link Bool ->
              (StabilizableClient a s, NullNetwork) ->
              Template (NullNetwork,
                        PeriodicClient (StablePkt a s))
stabilizer dim leader = \(client,network) ->
   template
      active := array' (1,dim) False
      count  := array' (1,dim) 0
      phase  := array' (1,dim) Snap
   in let
      tick i = action
          active!i := True

      poll i = request
         if active!i then
            if leader!i then
               return (Just (RequestPkt (count!i) (phase!i)))
            else
               if phase!i == Reset then
                  client.local_reset i
               s <- client.current_state i
               active!i := False
               phase!i := Snap
               return (Just (ResponsePkt (count!i) (phase!i) s))
         else
            m <- client.poll i
            return (map DataPkt m)

      receive i (DataPkt m) = client.receive i m

      receive i (RequestPkt c p) = action
         if not (leader!i) then
            active!i := True
            count!i  := c
            phase!i  := p

      receive i (ResponsePkt c p s) = action
         if active!i && leader!i && count!i == c then
            count!i := (count!i + 1) `mod` 4
            if phase!i == Snap then
```

```
            ok <- client.check_invariants i s
            if ok then
               active!i := False
            else
               phase!i := Reset
         else
            client.local_reset i
            active!i := False

   in (struct ..NullNetwork, struct ..PeriodicClient)
```

We do not intend to describe the actual protocol defined here in its entirety, since our focus in this case study is on the programming language itself. For a detailed coverage of the underlying algorithm, see [Var93]. Still, a few comments regarding the code might be in place.

- The `leader` parameter is an array which is supposed to tell whether the current node is the leader of a particular link subsystem. This value must of course be adapted to the actual location in the network where the current node is running; furthermore, it is assumed that this parameter, as well as other constants, are not subject to corruption.
- The `array'` initializer creates an array where all elements are initialized to a common value. Notice the extensive use of the array update syntax that O'Haskell provides.
- The interaction between methods `tick`, `poll`, and `receive` clarifies the snapshot and reset phases just outlined in [APSV91]. Most details, especially those regarding counter-flushing, are taken from [Var93]. The idea to use timer-based checking is taken from a partial implementation of the algorithm described in [LPS92].
- For comparison, the specification of this stabilizing scheme in terms of IO automata can be found in appendix C. The original, stabilizable node automaton is shown in figure C.2, and the stabilized node automaton in figures C.3 and C.4. It can be noticed that in addition to the lack of modularity in this formulation, the IOA code must also go to some trouble in order to emulate an O'Haskell-style "edge-triggered" semantics of action invocation where this is actually desired (note especially the role of the flag array $free_u$ in figure C.3).

4.5 Application: A global reset protocol

In this section we will show an example implementation of a protocol that is made self-stabilizing by means of the stabilizer protocol, as it is defined in the previous section. The protocol in question is a *global network reset* protocol taken from [APSV91], where it serves the same purpose as here.

Global reset should not be confused with the local correction phase of the stabilizer protocol, which is sometimes called *local reset* in the literature. It is also important to keep in mind that the protocol in this section is merely a user of the self-stabilizing services provided by the stabilizer protocol, even though it is itself a protocol extension that intercepts messages to and from *its* upper layers. In our terminology, global reset constitutes a *client* to the protocol layer below (i.e. the stabilizer, or the UDL layer in a non-stabilizing setting), and a *network* when seen from the protocol above it (which can be any protocol that occasionally may need to reset its peers on network in a controlled way).

The code for the global reset protocol can be found in appendix D. We refrain from giving any comments on its algorithmic details, the reader is referred to [APSV91] for that purpose. A few notes regarding actual coding issues is however called for:

- A reset-signal may originate from both the unknown client of this protocol (if the client detects the need for a global reset), or from an incoming reset message (if some other node has requested a global reset). This overloaded, but related use of method name `reset` is easily expressed in O'Haskell using records and subtyping.

- Since outgoing data is queued in this layer, the reset protocol can present (a subtype of) the `Network` interface to its client; consequently, the client does not need to be a subtype of `PolledClient`.

- For correct operation, incoming messages must be buffered during a reset phase. In the original IOA formulation (not shown in appendix C), emptying of incoming buffers (the `buffer` array in our encoding) is enabled whenever mode is Ready (remember that in the IOA model, action triggering can be gated by a static precondition). In O'Haskell, client reception of incoming data must be explicitly triggered, hence the buffers are flushed when mode *becomes* Ready (last three lines of procedure `reset_completed`).

- The last three methods defined, `current_state`, `check_invariants`, and `local_reset`, are only needed in order to make the protocol stabilizable. If the implementation was not intended to be used in conjunction with the stabilizing layer, these methods could have been omitted.

4.6 Putting it all together

Since the stabilizing extension from section 4.4 is a self-contained protocol layer, all that is needed in order to create a self-stabilizing reset protocol for some node is to connect the two layers using the stacking combinator:

```
dim                = 5
leader             = array' (1,dim) True
```

```
stable_global_reset =   global_reset dim
                              <||>
                      stabilizer dim leader
```

The result is a compound protocol which can be given as a client to a UDL, connected to a timer via its `tick` method, and used as a resettable network by any protocol that is a `ResetClient`. This is all visible in the inferred type:

```
stable_global_reset ::
   (ResetClient a, NullNetwork) ->
   Template (ResetNetwork,
             PeriodicClient (StablePkt (ResetMsg a) LocalState))
```

Thanks to subtyping of interfaces, the following protocol stacks are also configurable:

```
stack1 = stable_global_reset <||> udl dim
stack2 = global_reset dim <||> udl dim
stack3 = stabilizer dim leader <||> udl dim
```

Note, though, that we have avoided the question of how the `tick` method of the stabilizer protocol is connected to an external timer in these examples. In general, initializing a timer is not possible without triggering an action of some sort, so if we want to be able to do this while still using the stacking combinator, it must be generalized to the type of *commands* instead of templates. We have not done so here in order to keep the presentation simple, but our full implementation of the protocol actually utilizes Haskell-style overloading for this purpose, so that a single definition of the stacking combinator may cater for all conceivable uses.

4.7 Inheritance revisited

If the coding exercise of this chapter had been performed in an established object-oriented language like Java and C++, it is unlikely that the programmer had chosen to *modify* the code of an existing, non-stabilizable protocol like global reset in order to make it amenable to self-stabilization. Instead the spontaneous way to define the required methods `current_state`, `check_invariants`, and `local_reset` would probably be to implement them in a separate subclass that is related to the original protocol code by means of implementation inheritance.

As we have discussed in section 2.7, this style of programming is not supported in O'Haskell. Objects are closed worlds and state-variables are not accessible outside an object, so there is no automatic way of extending a given template with state-dependent code unless this has explicitly been prepared for by means of parameterization. However, in our particular case it appears like *modifying* instead of *inheriting* code is just a stylistic difference, since in order to even write down the implementation of a method like `local_reset`,

the programmer must already have gained access to (not to mention gained a full understanding of) the original source code anyway. We have already argued that this is likely to be the general pattern when the common examples of inheritance-as-parameterization have been taken aside, and we repeat our claim from section 2.7 that the lack of an inheritance mechanism in O'Haskell is acceptable, especially when the semantic complexity that is avoided by this restriction is taken into consideration.

On the other hand, once we have a client protocol that is stabilizable, the `stabilizer` implementation in th case study shows how we can (1) construct a new interface that has all the methods of the client plus a new one (`tick`), and (2) "override" the old method bodies with new code that occasionally makes calls to the old method implementations (`client.poll` and `client.receive`). Indeed this is a form of inheritance (inheritance by delegation, or *redirection*, as it should be called according to some definitions [AC96]), although we are probably much more used to think of this kind of program composition in other terms. However, when combined with the subtyping system of O'Haskell, inheritance by delegation constitutes an interesting programming technique, worthy of further exploration.

Our only main concern regarding this programming style is that delegating a large number of method calls to a private object might be a tedious exercise if some syntactic support for this style is not provided. Record stuffing can be seen as one half of such an aid, but a corresponding construct that "opens up" the scope captured in a record value must probably also be devised. We are currently investigating how this can best be done in O'Haskell.

4.8 Conclusions

In this chapter we have shown how a non-trivial network protocol, or more precisely, a *protocol transformation scheme*, is transcribed from an abstract notation suitable for formal reasoning, into a concrete program written in programming language O'Haskell. The exercise has been relatively painless, which we take as an indication of the merits of O'Haskell as a network programming language. We summarize our conclusions from this undertaking in the following paragraphs.

- O'Haskell is expressive enough to allow faithful transcription of both concurrency issues in the form of IOA notation, and informal mathematical language (c.f. especially the encoding of local predicates in the global reset protocol). Our impression is that one does not lose much in clarity and succinctness by going from an abstract formalism to a concrete implementation in O'Haskell, at least not in this particular case. We have not checked whether the formal proofs of self-stabilization still go through as easily using O'Haskell notation, though.

- The higher-order nature of O'Haskell allows the original protocol transformation scheme to be replaced by code that is parameterized over an

unknown protocol. This modular construction actually leads to *increased* clarity in the concrete implementation, compared to the original definition.

- Imperative constructs like loops, assignments, and array updates are as easy to use in O'Haskell as they are in the informal imperative notation of IOA, despite the fact that O'Haskell retains the property of being a *pure* functional language.

- The ability to separate protocol interfaces in an upper and and a lower part makes specification and implementation of protocol stacks both natural and flexible. The convention we use also blends well with the intuition that a more refined (smaller) interface plug should fit into the hole of less demanding (larger) socket.

- Type inference works perfectly well in this quite realistic example, in spite of an incomplete inference algorithm and several examples of non-trivial interaction between polymorphism and subtyping. In fact, all type annotations given in this text are completely superfluous, and serve only as a reading aid.

Chapter 5

Subtyping

In this chapter we will begin our formal development of O'Haskell, which will involve the polymorphic type system, the subtyping logic, and the type inference algorithm (this chapter), followed by the dynamic semantics of reactive objects in chapter 6. The formal treatment will for natural reasons not encompass the full language with all its different forms of syntactic abbreviations. Instead we will follow established practice and concentrate on a core calculus that is just expressive enough to demonstrate the central properties of the language. Specifically, we will leave out any coverage of general pattern matching, list comprehensions, and overloading in this treatment; not because these issues are trivial or unimportant in practice, but because they are already well covered in the literature on Haskell and elsewhere [Pet97, PJ87, Jon93, HHJW94, Aug87]. This modular approach is facilitated by the fact that the listed features are virtually orthogonal to the subtyping and concurrency extensions we will concentrate on in these chapters.

Another simplification we will make is to limit the formal treatment to a system with only first-order polymorphism. Haskell has employed higher-order polymorphism for quite some time, and indeed our current implementation also supports this feature, albeit as a truly experimental option. However, a higher-order treatment of subtyping raises some technical issues, for example regarding sub-kinding [Ste97], whose solution in the context of partial type inference will have to be reconsidered on basis of more practical experimentation. Moreover, a generalization of the current presentation to a higher-order formalism would also unnecessarily obscure the basic ideas of our type inference approach, which are largely independent of the distinction in order. A full coverage of higher-order subtyping in O'Haskell is therefore submitted to future work, although we will return to this issue and the technical questions it raises when we discuss the current implementation in chapter 7.

This chapter is organized as follows. Section 5.1 introduces the basic calculus and its type system, the subtyping logic, and a small extension that models programmer-defined records and datatypes. The basic properties of typing judgements for this calculus are stated and proved in section 5.2. In section 5.3

Term language:

$$
\begin{array}{rcll}
e & ::= & x & \textit{variable} \\
 & | & e\ e' & \textit{application} \\
 & | & \lambda x \rightarrow e & \textit{abstraction} \\
 & | & \textbf{let}\ x = e\ \textbf{in}\ e' & \textit{local definition}
\end{array}
$$

Type language:

$$
\begin{array}{rcll}
\tau, \rho & ::= & \alpha \mid t\ \tau_1 \ldots \tau_{n_t} & \textit{types} \\
\sigma & ::= & \forall \overline{\alpha}.(\tau | C) & \textit{type schemes} \\
C, D & ::= & \{\tau_i \leq \rho_i\}^i & \textit{constraint sets} \\
\Gamma & ::= & \{x_i : \sigma_i\}^i & \textit{assumptions}
\end{array}
$$

Figure 5.1: The core language

the inference algorithm is described and its theoretical properties are investigated. Section 5.4 contains a more practical exploration of the algorithm by means of some small inference examples, before the chapter ends in section 5.5 with a discussion on how type checking in the presence of incomplete type inference is addressed.

5.1 Type system

The starting point of our core calculus is the let-polymorphic type system of ML (a.k.a. the Hindley/Milner system), extended with subtype constraints, a subtype relation, and a subsumption rule. Such extensions are well represented in the literature [Kae92, Smi94, AW93, EST95]; in this paper we will adopt a formulation by Henglein [Hen96], which has the interesting merit of being "generic" in its subtyping theory. Since our system is obtained by "instantiation" with a particular form of subtyping theory, the results of [Hen96] directly carry over to our case. The generic part of this type system is shown in figures 5.1 and 5.2.

In our formulation, type variables are ranged over by α and β, while type constants are ranged over by t and s. The *arity* of a type constant t is denoted n_t. Type constants are considered drawn from some finite set that contains at least the function type constructor ($\rightarrow$) with arity 2. Substitutions are ranged over by Φ and Ψ. Terms and type schemes are considered equivalent up to α-conversion, and we will assume that terms have been renamed so that all bound variables are distinct and not equal to any free variables. For notational

$$\frac{}{C, \Gamma \cup \{x : \sigma\} \vdash_P x : \sigma}\ \textsc{TypVar}$$

$$\frac{C, \Gamma \vdash_P e : \tau' \to \tau \quad C, \Gamma \vdash_P e' : \tau'}{C, \Gamma \vdash_P e\ e' : \tau}\ \textsc{TypApp}$$

$$\frac{C, \Gamma \cup \{x : \tau'\} \vdash_P e : \tau}{C, \Gamma \vdash_P \lambda x \to e : \tau' \to \tau}\ \textsc{TypAbs}$$

$$\frac{C, \Gamma \vdash_P e : \sigma \quad C, \Gamma \cup \{x : \sigma\} \vdash_P e' : \tau}{C, \Gamma \vdash_P \textbf{let}\ x = e\ \textbf{in}\ e' : \tau}\ \textsc{TypLet}$$

$$\frac{C \cup D, \Gamma \vdash_P e : \tau \quad \overline{\alpha} \notin fv(C, \Gamma)}{C, \Gamma \vdash_P e : \forall \overline{\alpha}.\tau | D}\ \textsc{TypGen}$$

$$\frac{C, \Gamma \vdash_P e : \forall \overline{\alpha}.\tau | D \quad C \vdash_P [\overline{\tau}/\overline{\alpha}]D}{C, \Gamma \vdash_P e : [\overline{\tau}/\overline{\alpha}]\tau}\ \textsc{TypInst}$$

$$\frac{C, \Gamma \vdash_P e : \tau \quad C \vdash_P \tau \leq \tau'}{C, \Gamma \vdash_P e : \tau'}\ \textsc{TypSub}$$

Figure 5.2: The basic type system

convenience, we will also make use of the following abbreviations:

$$\begin{array}{rcl} \tau \to \rho & \equiv & (\to)\ \tau\ \rho \\ \tau | D & \equiv & \forall \emptyset.\tau | D \\ \tau & \equiv & \tau | \emptyset \\ C \vdash_P D & \equiv & C \vdash_P \tau \leq \rho \text{ for all } \tau \leq \rho \in D \\ \overline{\alpha} \notin S & \equiv & \alpha \notin S, \text{ for all } \alpha \in \overline{\alpha} \end{array}$$

It can easily be verified that if $C, \Gamma \vdash_P e : \forall \overline{\alpha}.\tau | D$ is derivable without the use of rule TypSub, and if all type schemes in Γ have empty constraint sets, then $\emptyset, \Gamma \vdash_P e : \forall \overline{\alpha}.\tau$ is derivable without rule TypSub and with empty constraint sets throughout (i.e. the judgement is also derivable in the original Hindley/Milner type system). Let $\Gamma \vdash^{HM} e : \sigma$ denote such a derivation. In section 5.3 we will prove a completeness result w.r.t. derivations of this limited form.

5.1.1 Subtype relation

Figure 5.3 shows the inference rules for our subtype relation. With the exception of rules SubDepth and SubConst, this definition directly corresponds to the one in [Hen96].

$$\frac{}{C \cup \{\tau \leq \rho\} \vdash_P \tau \leq \rho} \textsc{SubHyp}$$

$$\frac{}{C \vdash_P \tau \leq \tau} \textsc{SubRefl}$$

$$\frac{C \vdash_P \tau \leq \rho \quad C \vdash_P \rho \leq \tau'}{C \vdash_P \tau \leq \tau'} \textsc{SubTrans}$$

$$\frac{(C \vdash_P \tau_i \leq \rho_i)^{i \in t^+} \quad (C \vdash_P \rho_j \leq \tau_j)^{j \in t^-}}{C \vdash_P t\ \tau_1 \ldots \tau_{n_t} \leq t\ \rho_1 \ldots \rho_{n_t}} \textsc{SubDepth}$$

$$\frac{\tau <_P \rho}{C \vdash_P \Phi\tau \leq \Phi\rho} \textsc{SubConst}$$

Figure 5.3: Subtype relation

Rule SubDepth depends on the given *variance* of a type constant, as captured by the following notation: For every type constant t, let the sets $t^+, t^- \subseteq \{1 \ldots n_t\}$ represent the argument indices that the subtype relation should treat as co- and contravariant, respectively.

For a built-in type, these choices must of course be consistent with the dynamic semantics of terms of that type, which for the function symbol means that $(\rightarrow)^+ = \{2\}$, and $(\rightarrow)^- = \{1\}$ (see [CW85]). Thus, rule SubDepth replaces and generalizes the standard rule for functions (called Arrow in [Hen96]). As regards types defined by the programmer, section 5.1.2 will describe how variance information can be extracted from record and datatype declarations.

Our basic subtyping theory is defined in terms of polymorphic *subtype axioms* that may be instantiated by substitution, as witnessed by rule SubConst in figure 5.3.

Definition 5.1.1 (Subtype axiom). If $t\ \overline{\tau}$ and $s\ \overline{\rho}$ are ground type expressions, $t \neq s$, and $\overline{\tau}$ and $\overline{\rho}$ contain no occurrences of t and s, then $(t\ \overline{\tau} < s\ \overline{\rho})$ is a *subtype axiom* relating t and s.

We consider subtype axioms equivalent up to renaming of variables. To assert the existence of some subtype axiom, we write $(\tau < \rho) \in S$, or more conveniently $\tau <_S \rho$, where S is some given set.

The interaction between multiple subtype axioms is controlled by the following definition:

Definition 5.1.2 (Subtyping theory). A *subtyping theory* P is a finite set of subtype axioms such that

1. all axioms in P relate distinct pairs of type constants.

2. if $t\ \overline{\tau} <_P s\ \overline{\rho}$ and $s\ \overline{\rho}' <_P t'\ \overline{\tau}'$, then $\overline{\rho}$ and $\overline{\rho}'$ can be unified and $\Phi(t\ \overline{\tau} <_P t'\ \overline{\tau}')$, where Φ is a most general unifier of $\overline{\rho}$ and $\overline{\rho}'$.

Since the set of valid subtyping judgements depends on a particular subtyping theory P, we let the inference systems in figures 5.2 and 5.3 be parametric in P. This P has a role analogous to the partial order on types that is assumed in [Hen96]. To see that our definition is an instance of the latter, note that for each P the relation

$$\{\Phi(\tau, \rho) \mid \tau <_P \rho \wedge \Phi \text{ is a substitution}\}$$

contains all relationships derivable by our SubConst rule, and that its reflexive closure defines a partial order on ground type expressions as required. It is also trivial to verify that our formulation preserves the property that subtyping judgements are closed under substitution.

Our requirements on a subtyping theory are very liberal, and allow both multiple sub- and supertypes (*multiple inheritance* in object-oriented terms), and a form of rank-2 polymorphism that comes from the ability to use some type variables on just one side in a subtype axiom. In concrete programming terms, a subtyping theory expresses both the subtype relationships that our language provides as built-in (e.g. $Action < O\ \rho\ ()$), as well as the relationships that result from incremental type definitions made by the programmer (e.g. $CPoint < Point$, or $Ord\ \alpha < Eq\ \alpha$). Incremental definitions of records and datatypes will be described in the next subsection.

5.1.2 Records and datatypes

We now extend our core calculus to include programmer-defined records and datatypes. Because we have chosen to work with name inequivalence, no new subtyping rules need to be introduced here. The focus in this subsection will instead be on the kind of subtyping theory a particular set of type declarations defines.

We extend our term language with the following constructs:

e	::=	k	*datatype constructor*
	|	$\{k_i \to e_i\}^i$	*datatype selection*
	|	l	*record selector*
	|	$\{l_i = e_i\}^i$	*record construction*
	|	$\ldots$	

The datatype selection syntax might seem a little unusual, since it does not contain any expression to scrutinize. We have chosen the given formulation in order to emphasize the symmetry between datatype selection and record construction. The standard syntax for datatype selection, **case** e **of** $\{k_i \to e_i\}^i$, can be seen as syntactic sugar for the application $\{k_i \to e_i\}^i\ e$ in this calculus. A similar argument applies to the dot-notation used for record selection, i.e. $e.l \equiv l\ e$.

It should also be noted that the datatype selection expression matches constructor names with *lambda-abstractions*, that take exactly one argument standing for the *single* component of a datatype constructor. This choice has been made in order to simplify the calculus, although it is also a natural consequence of the record/datatype symmetry we wish to illustrate. The full generality of constructors with any arity can easily be encoded using *tuples*, which in turn can be adequately modeled in our calculus as a set of predefined type and term constants.

A *program* is defined as an expression within the scope of a set of top-level record and datatype declarations:

$$\begin{array}{rcl} p & ::= & \{d_i\}^i\ e \\ d & ::= & \textbf{struct}\ t\ \overline{\alpha} < \{s_i\ \overline{\rho}_i\}^i = \{l_j : \tau_j\}^j \\ & | & \textbf{data}\ t\ \overline{\alpha} > \{s_i\ \overline{\rho}_i\}^i = \{k_j\ \tau_j\}^j \end{array}$$

Here the s_i are called *base types*, which should be read as *supertypes* in the case of records, and *subtypes* in datatype declarations. We impose the natural restriction that the former types be only record types, and the latter types include only datatypes. The arity n_t of a declared type constant t is the length of the corresponding argument vector $\overline{\alpha}$. It is tacitly assumed that all type expressions respect arities. We furthermore make it an implicit precondition that all declared type constants, as well as all introduced record selectors and datatype constructors, are globally unique.

Now, for every declared record type $\textbf{struct}\ t\ \overline{\alpha} < \{s_i\ \overline{\rho}_i\}^i = \{l_j : \tau_j\}^j$, we define the following:

1. A type scheme for each selector

$$\sigma_{l_j} = \forall\overline{\alpha}.t\ \overline{\alpha} \to \tau_j$$

2. A set of subtype axioms

$$P_t = \{t\ \overline{\alpha} < s_i\ \overline{\rho}_i\}^i \cup \bigcup_i \{t\ \overline{\alpha} < [\overline{\rho}_i/\overline{\beta}](s'\ \overline{\tau}) \mid (s_i\ \overline{\beta} < s'\ \overline{\tau}) \in P_{s_i}\}$$

3. A selector closure

$$\widehat{t} = \{l_j\}^j \cup \bigcup_i \widehat{s_i}$$

4. A selector environment

$$\Pi_{\widehat{t}} = \{l_j : \sigma_{l_j}\}^j \cup \bigcup_i \{l : \forall\overline{\alpha}.t\ \overline{\alpha} \to [\overline{\rho}_i/\overline{\beta}]\tau \mid (l : \forall\overline{\beta}.s_i\ \overline{\beta} \to \tau) \in \Pi_{\widehat{s_i}}\}$$

Similarly, for every datatype declaration $\textbf{data}\ t\ \overline{\alpha} > \{s_i\ \overline{\rho}_i\}^i = \{k_j\ \tau_j\}^j$ we define:

1. A type scheme for each constructor

$$\sigma_{k_j} = \forall\overline{\alpha}.\tau_j \to t\ \overline{\alpha}$$

2. A set of subtype axioms

$$P_t = \{s_i\ \overline{\rho}_i < t\ \overline{\alpha}\}^i \cup \bigcup_i \{[\overline{\rho}_i/\overline{\beta}](s'\ \overline{\tau}) < t\ \overline{\alpha} \mid (s'\ \overline{\tau} < s_i\ \overline{\beta}) \in P_{s_i}\}$$

3. A constructor closure

$$\widehat{t} = \{k_j\}^j \cup \bigcup_i \widehat{s_i}$$

4. A constructor environment

$$\Sigma_{\widehat{t}} = \{k_j : \sigma_{k_j}\}^j \cup \bigcup_i \{k : \forall\overline{\alpha}.[\overline{\rho}_i/\overline{\beta}]\tau \to t\ \overline{\alpha} \mid (k : \forall\overline{\beta}.\tau \to s_i\ \overline{\beta}) \in \Sigma_{\widehat{s_i}}\}$$

We say that a set of declarations introducing type constants t_i is *well-formed* iff

- $\bigcup_i P_{t_i}$ is a valid subtyping theory.
- All $\widehat{t_i}$ are globally unique.[1]

. In the sequel we will tacitly assume that all expressions are typed in the scope of a well-formed set of type declarations, and that $P \supseteq \bigcup_i P_{t_i}$.

For both the record and datatype declaration forms, we let the variance information t^+ and t^- be the smallest sets of type parameter indices such that

$$C \vdash_P \Phi(t\ \overline{\alpha}) \leq \Phi'(t\ \overline{\alpha})$$

implies

$$C \vdash_P \Phi(s_i\ \overline{\rho}_i) \leq \Phi'(s_i\ \overline{\rho}_i)$$

and

$$C \vdash_P \Phi\tau_j \leq \Phi'\tau_j$$

for all i,j,Φ, and Φ', and all C such that (C, P) is *unambiguous*. Unambiguity will be formally defined in relation to the subject reduction theorem of the next section.

Computing these variance sets on basis of variances for the predefined type constants is straightforward; the only complication being that type declarations may be mutually recursive, and hence an iterative procedure will generally be

[1] In the full language we actually allow ambiguous $\widehat{t_i}$, for the convenience of the programmer. However, since we insist on name-inequivalence, such a system must necessarily lack the principal type property, hence type annotations are generally needed when this feature is used.

$$\frac{}{C, \Gamma \vdash l : \sigma_l}\ \textsc{TypSel}$$

$$\frac{}{C, \Gamma \vdash k : \sigma_k}\ \textsc{TypCon}$$

$$\frac{\Pi_{\{l_i\}^i} = \{l_i : \forall \overline{\alpha}.\tau \to \tau_i'\}^i \quad C, \Gamma \vdash_P e_i : [\overline{\rho}/\overline{\alpha}]\tau_i'}{C, \Gamma \vdash \{l_i = e_i\}^i : [\overline{\rho}/\overline{\alpha}]\tau}\ \textsc{TypStruct}$$

$$\frac{\Sigma_{\{k_i\}^i} = \{k_i : \forall \overline{\alpha}.\tau_i' \to \tau\}^i \quad C, \Gamma \vdash_P e_i : [\overline{\rho}/\overline{\alpha}]\tau_i' \to \tau'}{C, \Gamma \vdash_P \{k_i \to e_i\}^i : [\overline{\rho}/\overline{\alpha}]\tau \to \tau'}\ \textsc{TypAlts}$$

Figure 5.4: Records and datatypes

called for. The problem is however easy to formulate in terms of *abstract interpretation* of the given declarations, using an abstract domain that consists of the four subsets of $\{+, -\}$.

Note that since all constructor names k_j are required to be globally unique, there is no way of *modifying* the type of a constructor when a datatype is extended. Thanks to subsumption, old constructors can nonetheless be used to construct values of the new type. Likewise, for record types it is the subsumption rule that will allow us to apply old selectors to values of the new record type. The typing rules for records and datatypes are given in figure 5.4. Here (in rules TypStruct and TypAlts), as well as in the following development, the premises $\Pi_{\{l_i\}^i} = \ldots$ and $\Sigma_{\{k_i\}^i} = \ldots$ should be read as an assertion that corresponding record/selection expression is *well-formed*, i.e. that $\{l_i\}^i = \widehat{t}$ or $\{k_i\}^i = \widehat{t}$ for some t.

We have assumed here that both kinds of type declarations only mention variables that appear in the argument list $\overline{\alpha}$. As has been described by Läufer, Jones, and others, lifting this restriction for a constructor type naturally leads to a system with support for local existential quantification, while the corresponding generalization for selector types is best interpreted as local universal quantification [Läu92, Jon96b, Jon97]. Taking this idea further, by also letting the declared sub-/supertypes contain free variables, opens up some interesting possibilities to explore (recall that the generated subtype axioms remain valid in spite of this change). For one thing, a term can now be assigned an existentially quantified type simply by means of a type annotation, or even just by using the term in the right context. However, although this additional feature is natural and carries no extra implementational cost, it is not clear how useful it really is in practice.

5.2 Properties of typing judgements

In this section we state the main technical properties of our type system. This exposition will essentially be a recapitulation of results that are proven in general in [Hen96], and which continue to hold for our basic system in particular. Furthermore, our record and datatype extension only trivially affects most of the original proofs. We will only show the necessary supplements when this is not the case.

Lemma 5.2.1. *If* $C, \Gamma \vdash_P e : \sigma$ *then* $\Phi C, \Phi\Gamma \vdash_P e : \Phi\sigma$ *for all substitutions* Φ.

Lemma 5.2.2. *If* $C, \Gamma \vdash_P e : \sigma$ *and* $D \vdash_P C$ *then* $D, \Gamma \vdash_P e : \sigma$.

Proposition 5.2.3. *Let* $\overline{\alpha} \notin fv(C, \Gamma)$. *Then* $C, \Gamma \vdash_P e : \forall\alpha.\tau|D$ *iff* $C \cup D, \Gamma \vdash_P e : \tau$.

For the purpose of the next statement, the subtype relation in figure 5.3 is extended to type schemes as follows:[2]

1. $C \vdash_P \sigma \leq \tau$ if $C, \{x : \sigma\} \vdash_P x : \tau$.
2. $C \vdash_P \sigma \leq \sigma'$ if for all D and τ such that $D \vdash_P C$ and $D \vdash_P \sigma' \leq \tau$ we have $D \vdash_P \sigma \leq \tau$.

Proposition 5.2.4 (Invariance under subtyping). *Let* $C \vdash_P \sigma \leq \sigma'$. *Then:*

1. *If* $C, \Gamma \vdash_P e : \sigma$ *then* $C, \Gamma \vdash_P e : \sigma'$.
2. *If* $C, \Gamma \cup \{x : \sigma'\} \vdash_P e : \sigma''$ *then* $C, \Gamma \cup \{x : \sigma\} \vdash_P e : \sigma''$.

Theorem 5.2.5 (Principal types). *Let* $fv(e) \subseteq dom(\Gamma)$, *and let all record and datatype selection expressions in* e *be well-formed. Then there exists a* σ *such that:*

1. $\emptyset, \Gamma \vdash_P e : \sigma$[3]
2. *for all* σ', *if* $\emptyset, \Gamma \vdash_P e : \sigma'$ *then* $\emptyset \vdash_P \sigma \leq \sigma'$.

Proof. Follows from the completeness of Henglein's inference algorithm W (see theorem 2.16 in [Hen96]). We will only show that completeness still holds in spite of our extensions.

[2] Since our subtype axioms may relate types containing arbitrary sets of variables in their arguments, we have reasons to avoid Henglein's restriction on the free variables in the smaller scheme. No fundamental difficulties arise because of this relaxation; in particular, the TypGen case of proposition 5.2.4, part 2, still goes through by the same argument as one uses in the proof of lemma 5.2.1.

[3] Note that this statement does not say anything about the *satisfiability* of the constraints that are generally contained in σ.

Add the following clauses to the definition of W (figure 3 in [Hen96]):

$$\begin{aligned}
W(\Gamma, l) &= \textbf{let } \forall\overline{\alpha}.\tau = \sigma_l \textbf{ in } (\emptyset, [\overline{\nu}/\overline{\alpha}]\tau) \\
W(\Gamma, k) &= \textbf{let } \forall\overline{\alpha}.\tau = \sigma_k \textbf{ in } (\emptyset, [\overline{\nu}/\overline{\alpha}]\tau) \\
W(\Gamma, \{l_i = e_i\}^i) &= \textbf{let } \{l_i : \forall\overline{\alpha}.\tau \to \tau_i'\}^i = \Pi_{\{l_i\}^i} \textbf{ in} \\
&\quad \textbf{let } (C_i, \rho_i) = W(\Gamma, e_i) \textbf{ in} \\
&\quad (\textstyle\bigcup_i C_i \cup \{\rho_i \leq [\overline{\nu}/\overline{\alpha}]\tau_i'\}^i, [\overline{\nu}/\overline{\alpha}]\tau) \\
W(\Gamma, \{k_i \to e_i\}^i) &= \textbf{let } \{k_i : \forall\overline{\alpha}.\tau_i' \to \tau\}^i = \Sigma_{\{k_i\}^i} \textbf{ in} \\
&\quad \textbf{let } (C_i, \rho_i) = W(\Gamma, e_i) \textbf{ in} \\
&\quad (\textstyle\bigcup_i C_i \cup \{\rho_i \leq [\overline{\nu}/\overline{\alpha}]\tau_i' \to \nu\}^i, [\overline{\nu}/\overline{\alpha}]\tau \to \nu)
\end{aligned}$$

We will show that if $C', \Phi\Gamma \vdash_P \{l_i = e_i\}^i : \tau'$, then $W(\Gamma, \{l_i = e_i\}^i)$ succeeds with (C, τ), and there is a Φ' such that $C' \vdash_P \Phi' C$, $\Phi\Gamma = \Phi'\Gamma$, and $C' \vdash_P \Phi'\tau \leq \tau'$. The case for datatype selection follows analogously, and the constructor and selector cases are trivial.

The given derivation must be of the form

$$\frac{\Pi_{\{l_i\}^i} = \{l_i : \forall\overline{\alpha}.\tau \to \tau_i'\}^i \quad C', \Phi\Gamma \vdash_P e_i : [\overline{\rho}/\overline{\alpha}]\tau_i'}{C', \Phi\Gamma \vdash_P \{l_i = e_i\}^i : [\overline{\rho}/\overline{\alpha}]\tau} \textsc{TypStruct}$$

By the induction hypothesis we have that $W(\Gamma, e_i)$ succeeds with (C_i, ρ_i), and that there are Φ_i' such that

$$\begin{aligned}
&C' \vdash_P \Phi_i' C_i \\
&\Phi\Gamma = \Phi_i'\Gamma \\
&C' \vdash_P \Phi_i'\rho_i \leq [\overline{\rho}/\overline{\alpha}]\tau_i'
\end{aligned}$$

Since the (C_i, ρ_i) can only have variables in Γ in common, and since all $\Phi_i'|_{fv(\Gamma)}$ are identical, we may assume that all Φ_i' commute. We also know that $fv(\tau_i, \tau) \subseteq \overline{\alpha}$, since selector environments have no free variables.

By assumption, $\{l_i = e_i\}^i$ is well-formed, hence $W(\Gamma, \{l_i = e_i\}^i)$ succeeds with $(\bigcup_i C_i \cup \{\rho_i \leq [\overline{\nu}/\overline{\alpha}]\tau_i'\}^i, [\overline{\nu}/\overline{\alpha}]\tau)$. Now let $\Phi' = \Phi_1' \circ \ldots \circ \Phi_n' \circ [\overline{\rho}/\overline{\nu}]$, and we have

$$\begin{aligned}
&C' \vdash_P \Phi'(\textstyle\bigcup_i C_i \cup \{\rho_i \leq [\overline{\nu}/\overline{\alpha}]\tau_i'\}^i) \\
&\Phi\Gamma = \Phi'\Gamma \\
&C' \vdash_P \Phi'[\overline{\nu}/\overline{\alpha}]\tau = [\overline{\rho}/\overline{\alpha}]\tau
\end{aligned}$$

as required. □

For the purpose of our subject reduction theorem, we define a reduction relation $\mapsto$ for our calculus in figure 5.5. As can be seen, the relation is quite general and encompasses both a call-by-name and a call-by-value semantics. With the exception of the axioms RedSel and RedCase, and the rules RedStruct and RedAlts, the same relation also appears in [Hen96].

We also need a notion of *unambiguity*, that allows us to draw certain natural conclusions about the way subtyping judgements are derived.

Reduction axioms:

$$
\begin{array}{rcll}
(\lambda x \to e)\ e' & \mapsto & [e'/x]e & \textsc{RedBeta} \\
\textbf{let}\ x = e\ \textbf{in}\ e' & \mapsto & [e/x]e' & \textsc{RedUnfold} \\
l_j\ \{l_i = e_i\}^i & \mapsto & e_j & \textsc{RedSel} \\
\{k_i \to e_i\}^i\ (k_j\ e) & \mapsto & e_j\ e & \textsc{RedCase} \\
e & \mapsto & e & \textsc{RedRefl} \\
\lambda x \to e\ x & \mapsto & e & \textsc{RedEta}
\end{array}
$$

Reduction rules:

$$\frac{e_1 \mapsto e_1' \quad e_2 \mapsto e_2'}{e_1\ e_2 \mapsto e_1'\ e_2'}\ \textsc{RedApp}$$

$$\frac{e \mapsto e'}{\lambda x \to e \mapsto \lambda x \to e'}\ \textsc{RedAbs}$$

$$\frac{e_1 \mapsto e_1' \quad e_2 \mapsto e_2'}{\textbf{let}\ x = e_1\ \textbf{in}\ e_2 \mapsto \textbf{let}\ x = e_1'\ \textbf{in}\ e_2'}\ \textsc{RedLet}$$

$$\frac{e_i \mapsto e_i'}{\{l_i = e_i\}^i \mapsto \{l_i = e_i'\}^i}\ \textsc{RedStruct}$$

$$\frac{e_i \mapsto e_i'}{\{k_i \to e_i\}^i \mapsto \{k_i \to e_i'\}^i}\ \textsc{RedAlts}$$

Figure 5.5: Reduction of terms

Definition 5.2.6 (Unambiguity). A constraint-set/subtyping-theory pair (C, P) is *unambiguous* iff the following conditions hold:

1. If $C \vdash_P t\ \tau_1 \ldots \tau_{n_t} \leq t\ \rho_1 \ldots \rho_{n_t}$ then $C \vdash_P \tau_i \leq \rho_i$ for all $i \in t^+$ and $C \vdash_P \rho_i \leq \tau_i$ for all $i \in t^-$.

2. For any two distinct record type constructors t and s, if $C \vdash_P t\ \overline{\tau} \leq s\ \overline{\rho}$ then there is a subtype axiom $t\ \overline{\alpha} <_P s\ \overline{\rho}'$ such that $C \vdash_P [\overline{\tau}/\overline{\alpha}](s\ \overline{\rho}') \leq s\ \overline{\rho}$.

3. For any two distinct datatype constructors t and s, if $C \vdash_P t\ \overline{\tau} \leq s\ \overline{\rho}$ then there is a subtype axiom $t\ \overline{\tau}' <_P s\ \overline{\alpha}$ such that $C \vdash_P t\ \overline{\tau} \leq [\overline{\rho}/\overline{\alpha}](t\ \overline{\tau}')$.

The following lemma states a form of sanity property for our record and datatype extension:

Lemma 5.2.7. *If $\{d_i\}^i$ is a well-formed set of type declarations introducing t_i, then $(\emptyset, \bigcup_i P_{t_i})$ is unambiguous.*

Proof. All conditions are proved by induction on the derivation of the given subtyping judgement. The only interesting cases concern conditions 2 and 3, in the specific case of rule SubTrans. We show the case for record types here, the datatype case follows analogously.

Assume $P = \bigcup_i P_{t_i}$. The given derivation must be of the form

$$\frac{\emptyset \vdash_P t\ \overline{\tau} \leq t'\ \overline{\tau}' \quad \emptyset \vdash_P t'\ \overline{\tau}' \leq s\ \overline{\rho}}{\emptyset \vdash_P t\ \overline{\tau} \leq s\ \overline{\rho}}\ \textsc{SubTrans}$$

We have three subcases depending on t':

Case ($t' = s$). By the induction hypothesis we already have $t\ \overline{\alpha} <_P s\ \overline{\rho}'$ and $\emptyset \vdash_P [\overline{\tau}/\overline{\alpha}](s\ \overline{\rho}') \leq s\ \overline{\tau}'$, and $\emptyset \vdash_P [\overline{\tau}/\overline{\alpha}](s\ \overline{\rho}') \leq s\ \overline{\rho}$ simply follows by transitivity.

Case ($t' = t$). By the induction hypothesis we have $t\ \overline{\alpha} <_P s\ \overline{\rho}'$ and $\emptyset \vdash_P [\overline{\tau}'/\overline{\alpha}](s\ \overline{\rho}') \leq s\ \overline{\rho}$. We also have that $\emptyset \vdash_P t\ \overline{\tau} \leq t\ \overline{\tau}'$, so from the definition of variance for t it follows that $\emptyset \vdash_P [\overline{\tau}/\overline{\alpha}](s\ \overline{\rho}') \leq [\overline{\tau}'/\overline{\alpha}](s\ \overline{\rho}')$. Hence we have $\emptyset \vdash_P [\overline{\tau}/\overline{\alpha}](s\ \overline{\rho}') \leq s\ \overline{\rho}$ by transitivity.

Case ($t' \notin \{t, s\}$). By the induction hypothesis we both have that $t\ \overline{\alpha} <_P t'\ \overline{\rho}'$ and $\emptyset \vdash_P [\overline{\tau}/\overline{\alpha}](t'\ \rho') \leq t'\ \overline{\tau'}$, as well as $t'\ \overline{\alpha}' <_P s\ \overline{\rho}''$ and $\emptyset \vdash_P [\overline{\tau'}/\overline{\alpha}'](s\ \overline{\rho}'') \leq s\ \overline{\rho}$. From the definition of a subtyping theory it follows that there also exists a subtype axiom $t\ \overline{\alpha} < [\overline{\rho}'/\overline{\alpha}'](s\ \overline{\rho}'')$ in P. Furthermore, since we have $\emptyset \vdash_P [\overline{\tau}/\overline{\alpha}][\overline{\rho}'/\overline{\alpha}'](t'\ \overline{\alpha}') \leq [\overline{\tau}/\overline{\alpha}](t'\ \overline{\alpha}')$, it follows from the definition of variance for t' that $\emptyset \vdash_P [\overline{\tau}/\overline{\alpha}][\overline{\rho}'/\overline{\alpha}'](s\ \overline{\rho}'') \leq [\overline{\tau}/\overline{\alpha}](s\ \overline{\rho}'')$. By transitivity we thus have $\emptyset \vdash_P [\overline{\tau}/\overline{\alpha}][\overline{\rho}'/\overline{\alpha}'](t'\ \overline{\alpha}') \leq s\ \overline{\rho}$ as required.

□

In other words, an unambiguous pair $(C, \bigcup_i P_{t_i})$ is definitely achievable, and is just a matter of choosing C so that its transitive closure does not interfere with the subtyping rules SubDepth and SubConst.

Theorem 5.2.8 (Subject reduction). *Let (C, P) be unambiguous. Then, if $C, \Gamma \vdash_P e : \sigma$ holds and $e \mapsto e'$, $C, \Gamma \vdash_P e' : \sigma$ holds as well.*

Proof. By induction on the derivation of $e \mapsto e'$. The proof is essentially that of theorem 3.5 in [Hen96], only supplemented with cases for the new reduction rules and axioms (note that Henglein's precondition to theorem 3.5 is implied by our definition of unambiguity). We show the case for RedSel; RedCase follows analogously, and the other rules are straightforward.

We have $l_j\ \{l_i = e_i\}^i \mapsto e_j$ and a typing derivation for $l_j\ \{l_i = e_i\}^i$. W.l.o.g. we may assume that the last step in this derivation is formed by TypApp, since any subsequent TypSub or TypGen steps may always be restored afterwards. Likewise, we may assume that the first premise of rule TypApp is

a TypSel/TypInst chain, since all necessary subtype matching between the subexpressions can be confined to a single subtyping step in the right premise. Thus we have

$$\dfrac{\dfrac{\dfrac{}{C,\Gamma \vdash_P l_j : \forall\overline{\beta}.s\ \overline{\beta} \to \rho}\ \textsc{TypSel}}{C,\Gamma \vdash_P l_j : s\ \overline{\rho} \to [\overline{\rho}/\overline{\beta}]\rho}\ \textsc{TypInst} \quad C,\Gamma \vdash_P \{l_i = e_i\}^i : s\ \overline{\rho}}{C,\Gamma \vdash_P l_j\ \{l_i = e_i\}^i : [\overline{\rho}/\overline{\beta}]\rho}\ \textsc{TypApp}$$

where the right premise is derived as follows:

$$\dfrac{\dfrac{\Pi_{\{l_i\}^i} = \{l_i : \forall\overline{\alpha}.t\ \overline{\alpha} \to \tau_i\} \quad C,\Gamma \vdash_P e_i : [\overline{\tau}/\overline{\alpha}]\tau_i}{C,\Gamma \vdash_P \{l_i = e_i\}^i : t\ \overline{\tau}}\ \textsc{TypStruct} \quad C \vdash_P t\ \overline{\tau} \leq\ s\ \overline{\rho}}{C,\Gamma \vdash_P \{l_i = e_i\}^i : s\ \overline{\rho}}\ \textsc{TypSub}$$

There are now two cases, depending on t.

Case ($t = s$). Then, by construction, $\overline{\beta} = \overline{\alpha}$ and $\tau_j = \rho$. Since we have $C \vdash_P t\ \overline{\tau} \leq\ t\ \overline{\rho}$, it follows from the definition of variance for t that we have $C \vdash_P [\overline{\tau}/\overline{\alpha}]\rho \leq\ [\overline{\rho}/\overline{\alpha}]\rho$ as well. Hence we can construct a derivation of $C,\Gamma \vdash_P e_j : [\overline{\rho}/\overline{\alpha}]\rho$, as required.

Case ($t \neq s$). By the assumption about unambiguity, there must exist a subtype axiom $t\ \overline{\alpha} <_P s\ \overline{\rho}'$ such that $C \vdash_P [\overline{\tau}/\overline{\alpha}](s\ \overline{\rho}') \leq\ s\ \overline{\rho}$. Then, by construction, τ_j must be equal to $[\overline{\rho}'/\overline{\beta}]\rho$. Now, since we have $C \vdash_P [\overline{\tau}/\overline{\alpha}][\overline{\rho}'/\overline{\beta}](s\ \overline{\beta}) \leq\ [\overline{\rho}/\overline{\beta}](s\ \overline{\beta})$ it follows from the definition of variance for s that $C \vdash_P [\overline{\tau}/\overline{\alpha}][\overline{\rho}'/\overline{\beta}]\rho \leq\ [\overline{\rho}/\overline{\beta}]\rho$. Hence we can construct a derivation of $C,\Gamma \vdash_P e_j : [\overline{\rho}/\overline{\beta}]\rho$, as required.

□

We could also go on here and establish a *well-typed-terms-cannot-go-wrong* result about our calculus. Such a result can be obtained by means of the subject reduction theorem, using the standard approach of letting erroneous terms (e.g. terms that attempt to apply a record as a function) reduce to a distinct value *wrong* that has no type. However, even though these steps are straightforward, they are a bit tedious, and we will not pursue them here.

5.3 The inference algorithm

We will now turn to the partial inference algorithm we consider our main result of this chapter. The core of this algorithm is an approximating subtype constraint solver, that has the merit of being simple and efficient — in fact it is defined as a small extension to the very efficient unification algorithm of Martelli and Montanari [MM82]. The main characteristic of our solver is that it approximates constraints of the form $\alpha \leq \beta$ as equality constraints, and resorts

to unification in these cases. When all such constraints are removed from a constraint set, computation of least upper bounds or greatest lower bounds for the remaining variables becomes straightforward, and the algorithm can continue (just like in ordinary unification) by solving any constraints induced by the arguments of the bounding type expressions.

This strategy is not very refined, however, and is bound to fail in certain situations. Consider the constraint set

$$\{\alpha \leq \mathit{CPoint}, \alpha \leq \beta, \mathit{Point} \leq \beta\}$$

With our strategy, α would be unified with β, resulting in the unsolvable constraint set $\{\alpha \leq \mathit{CPoint}, \mathit{Point} \leq \alpha\}$ (assuming $\mathit{CPoint} <_P \mathit{Point}$). In this case, unifying α with *CPoint* would have been a better way to proceed.

So, if we want to stick to our simple strategy (which we really do, considering its attractively close relationship to well-known unification techniques), it becomes vital to feed as small constraint sets as possible into the constraint solver, in order to minimize the "damage" that variable/variable constraints can give rise to. For example, if the constraint set above actually was generated as the union of the sets $\{\alpha \leq \mathit{CPoint}\}$ and $\{\alpha \leq \beta, \mathit{Point} \leq \beta\}$, solving the first set separately would lead to success even with our simple strategy.

This requirement actually puts us closer to the standard inference algorithm W [Mil78] than what is customary in the literature on subtype inference. As we have indicated, we will have good reasons for solving constraints as soon as they are generated, rather than propagating them upwards in the syntax tree as input to some final simplification/solving pass. The complication we run into is of course that constraints involving variables free in the assumption environment cannot be solved immediately if we want the output of the algorithm to be predictable. However, we will postpone the discussion on how we address this problem until we have presented the constraint solver.

5.3.1 Solving constraints

The definition of the constraint solver is given in figure 5.6. It is presented as an inference system for judgements on the form $\Phi \models_P C$, which should have the operational interpretation "given a constraint set C and subtyping theory P, return substitution Φ". If a constraint set does not match any of the five clauses, the result of the algorithm is considered to be FAILURE. The union symbol $\uplus$ used for pattern-matching against subsets serves to indicate that the subsets must be disjoint.

This algorithm, as well as the subsequent inference algorithm, depends crucially on the ability to generate fresh type variables. Instead of burdening the presentation with unessential details concerning name supplies, we use the following convention: the symbol ν always represents a type variable that is distinct from any other variable in the derivation or its context, except for other occurrences of ν in the same rule. Likewise, we let $\overline{\nu}$ represent a vector of zero or more unique variables, equal only to the variables denoted by other occurrences of $\overline{\nu}$ in the same rule.

$$\frac{}{[\,] \models_P \emptyset}\ \textsc{CTriv}$$

$$\frac{\Phi \models_P [\beta/\alpha]C}{\Phi \circ [\beta/\alpha] \models_P C \uplus \{\alpha \leq \beta\}}\ \textsc{CVar}$$

$$\frac{\Phi \models_P C \cup \{\tau_i \leq \rho_i\}^{i \in t^+} \cup \{\rho_j \leq \tau_j\}^{j \in t^-}}{\Phi \models_P C \uplus \{t\ \tau_1 \ldots \tau_{n_t} \leq t\ \rho_1 \ldots \rho_{n_t}\}}\ \textsc{CDepth}$$

$$\frac{\begin{array}{c}\alpha \notin fv(C, \overline{\tau}_i, \overline{\rho}_j) \qquad t = (\sqcup_P \overline{t_i}) \parallel (\sqcap_P \overline{s_j}) \\ \Phi \models_P C \cup \{t_i\ \overline{\tau}_i \leq t\ \overline{\nu}\}^i \cup \{t\ \overline{\nu} \leq s_j\ \overline{\rho}_j\}^j\end{array}}{\Phi \circ [t\ \overline{\nu}/\alpha] \models_P C \uplus \{t_i\ \overline{\tau}_i \leq \alpha\}^i \cup \{\alpha \leq s_j\ \overline{\rho}_j\}^j}\ \textsc{CMerge}$$

$$\frac{\begin{array}{c}t\ \overline{\tau}' <_P s\ \overline{\rho}' \qquad \overline{\alpha} = fv(\overline{\tau}', \overline{\rho}') \\ \Phi \models_P C \cup \{t\ \overline{\tau} \leq [\overline{\nu}/\overline{\alpha}](t\ \overline{\tau}')\} \cup \{[\overline{\nu}/\overline{\alpha}](s\ \overline{\rho}') \leq s\ \overline{\rho}\}\end{array}}{\Phi \models_P C \uplus \{t\ \overline{\tau} \leq s\ \overline{\rho}\}}\ \textsc{CSub}$$

Figure 5.6: The constraint solver

Clauses CTriv to CMerge in figure 5.6 essentially constitute the unification algorithm of Martelli and Montanari, or more precisely, their nondeterministic *specification* of the algorithm. We prefer to use this abstract formulation here because of its conciseness; the reader is referred to chapter 7 and then further on to [MM82] for some more concrete information on how to make an efficient implementation, especially on how to avoid the costly occurs-check in clause CMerge.

The computation of least upper bounds / greatest lower bounds in clause CMerge is of course added by us; the reader should note, though, that our formulation degenerates to the original one in case the trivial subtyping theory $P = \emptyset$ is given. These standard algorithms use a partial order on pure type constants derived from P, which relates t and s iff $t = s$ or $t\ \overline{\tau} <_P s\ \overline{\rho}$ for some $\overline{\tau}$ and $\overline{\rho}$. Since neither least upper bounds nor greatest lower bounds are guaranteed to exist (for one thing, $\overline{\tau}$ or $\overline{\rho}$ may be empty), both algorithms might return FAILURE.

The "fatbar" operator in $(\sqcup_P \overline{t_i}) \parallel (\sqcap_P \overline{s_j})$ picks one of its arguments in a failure-avoiding manner. In case both arguments evaluate successfully this choice is assumed to be guided by the inference algorithm, so that the alternative is taken which results in the smallest inferred type. If neither alternative is better than the other we somewhat arbitrarily specify that the left argument is selected. Decorating the constraint solver with an extra parameter represent-

ing the type that should be minimized is straightforward, so we leave out the details in the interest of brevity.

Clause CSub is our main extension to the unification algorithm; it has no correspondence in the original formulation since it handles the case where two distinct type constants are compared. Note that this rule is only applicable if there exists an appropriate subtype axiom in P.

The following lemma states a soundness property of the constraint solver.

Lemma 5.3.1. *If* $\Phi \models_P C$ *then* $\emptyset \vdash_P \Phi C$.

Proof. By induction on the derivation of $\Phi \models_P C$. □

Since the algorithm deliberately discards certain solutions, completeness cannot hold by definition. However, a distinctive characteristic of our approach is that it is a conservative extension of *unification*. This is made precise below.

Definition 5.3.2. We say that a substitution Φ *unifies* a constraint set C iff $\Phi\tau = \Phi\rho$ for all $\tau \leq \rho \in C$.

Lemma 5.3.3. *If* Φ *unifies* C*, then* $\Phi' \models_P C$ *and* $\Phi = \Phi'' \circ \Phi'$ *for some* Φ''.

Proof. By the same argument as in [MM82], noting that

1. In clause CMerge, all t_i and s_j must be equal, and since at least one of $\overline{t_i}$ and $\overline{s_j}$ must be non-empty, $(\sqcup_P \overline{t_i}) \between (\sqcap_P \overline{s_j})$ trivially succeeds.
2. Since a subtype axiom never relates a type constant to itself, clause CSub can never match.
3. The case where a type constant is *non-variant* (that is, $i \notin t^+ \cup t^-$ for some $i \leq n_t$) only makes the algorithm output *more* general than the unifying substitution.

□

A vital property of the constraint solver is that it terminates on all input, either with a substitution, or with the implicit result failure if there is no matching clause. Formally,

Proposition 5.3.4. *The relation* $\Phi \models_P C$ *is decidable.*

Proof. We show that there can be no infinite derivations of $\Phi \models_P C$ by considering the *size* $|C|$ of a constraint set C, as defined by

$$
\begin{array}{lcl}
|\{\tau_i \leq \rho_i\}^i| & = & \sum_i |\tau_i \leq \rho_i| \\
|\alpha \leq \tau| & = & |\tau| \\
|\tau \leq \alpha| & = & |\tau| \\
|t\ \overline{\tau_i} \leq t\ \overline{\rho_i}| & = & 1 + 2\sum_i |\tau_i \leq \rho_i| \\
|t\ \overline{\tau} \leq s\ \overline{\rho}| & = & 1 + |t\ \overline{\tau} \leq t\ \overline{\tau}'| + |s\ \overline{\rho}' \leq s\ \overline{\rho}| \\
 & & \text{if } t\ \overline{\tau}' <_P s\ \overline{\rho}' \\
|\overline{\tau_i}| & = & \sum_i |\tau_i| \\
|\alpha| & = & 1 \\
|t\ \overline{\tau_i}| & = & 4 + 2(|\overline{\tau_i}| + n_{\max} + k_t)
\end{array}
$$

where $n_{\max}$ is the maximal arity of any t, and k_t is computed for each t by examining all axioms in P of the form $(t\ \overline{\tau} < s\ \overline{\rho})$ or $(s\ \overline{\rho} < t\ \overline{\tau})$, and taking the sum of the sizes of each $\overline{\tau}$ and $\overline{\rho}$ thus encountered. The well-foundedness of this definition follows from the requirement that the type constants related by a subtype axioms occur just once in the axiom (see definition 5.1.1).

It is now straightforward to show that every premise in the definition of $\Phi \models_P C$ involves a constraint set of strictly lesser size than the constraint set of the corresponding conclusion. The crucial step is rule CMerge, where one expands the premise using rules CSub and CDepth, and then utilizes the fact that for every τ and ρ, $|\tau \leq \rho|$ is less than $|\tau| + |\rho|$. □

Another important aspect of the solver is that its result is independent of the nondetermistic choices that are allowed by its specification.

Proposition 5.3.5. *If the constraint solver fails to solve a constraint set C, then it cannot succeed. If it succeeds with a substitution Φ, then Φ is uniquely determined by C up to renaming.*

Proof. The inference system of fig 5.6 can actually be read as a rewrite system for constraint sets, where the inference step

$$\frac{\Phi' \models_P C'}{\Phi' \circ \Phi \models_P C}$$

corresponds to the rewrite step $C \overset{\Phi}{\longmapsto} C'$. We let this Φ be the identity substitution if it is not explicitly identified in a clause.

It can easily be shown by enumeration of possibilities that if $C \overset{\Phi_1}{\longmapsto} C_1$ and $C \overset{\Phi_2}{\longmapsto} C_2$, then $C_1 \overset{\Phi_1'}{\longmapsto} C'$ and $C_2 \overset{\Phi_2'}{\longmapsto} C'$, and there is a renaming Ψ such that $\Phi_1' \circ \Phi_1 = \Psi \circ \Phi_2' \circ \Phi_2$. The only interesting case is the combination of clauses CVar and CMerge, where the CMerge premise $\alpha \notin fv(C, \overline{\tau}_i, \overline{\rho}_i)$ is essential.

The previous property can now be lifted to arbitrary chains of rewrite steps using induction on the length of the combined chains. We have that if $C \overset{\Phi_1}{\longmapsto}{}^{*} C_1$ in m steps and $C \overset{\Phi_2}{\longmapsto}{}^{*} C_2$ in n steps, then $C_1 \overset{\Phi_1'}{\longmapsto}{}^{*} C'$ in n steps and $C_2 \overset{\Phi_2'}{\longmapsto}{}^{*} C'$ in m steps, and there is a renaming Ψ such that $\Phi_1' \circ \Phi_1 = \Psi \circ \Phi_2' \circ \Phi_2$.

Since failure of the constraint solver is equivalent to a constraint set for which there exists no rewrite step, the desired result follows as a corollary. □

5.3.2 Algorithm definition

The actual inference algorithm is shown in figure 5.7. Again we use a formulation in terms of an inference system, whose judgements $C, \Gamma \models_P e : \tau$ should be read "given an assumption environment Γ, a subtyping theory P, and an expression e, return type τ and constraint set C". We will assume here that Γ is of the form $\{x_i : \forall \overline{\alpha}_i.\tau_i\}$, since handling of constrained type schemes will not be meaningful in our setting until we introduce type annotations in section 5.5.

$$\frac{\overline{\beta_i} = fv(\tau) \quad C = \{\nu_i \leq \beta_i\}^{\beta_i \in (\tau^- \setminus \overline{\alpha})} \cup \{\beta_i \leq \nu_i\}^{\beta_i \in (\tau^+ \setminus \overline{\alpha})}}{C, \Gamma \cup \{x : \forall \overline{\alpha}.\tau\} \models_P x : [\overline{\nu_i}/\overline{\beta_i}]\tau} \textsc{InfVar}$$

$$\frac{C, \Gamma \models_P e : \tau \quad C_i, \Gamma \models_P e_i : \tau_i \quad \Phi \models_P \{\tau \leq \overline{\tau_i} \to \nu\}}{\Phi(C \cup \bigcup_i C_i), \Gamma \models_P e\ \overline{e_i} : \Phi\nu} \textsc{InfApp}$$

$$\frac{C, \Gamma \cup \{x_i : \nu_i\}^i \models_P e : \tau \quad \Phi \models_P C \backslash C_\Gamma}{\Phi(C_\Gamma), \Gamma \models_P \lambda \overline{x_i} \to e : \Phi(\overline{\nu_i} \to \tau)} \textsc{InfAbs}$$

$$\frac{C, \Gamma \models_P e : \tau \quad C', \Gamma \cup \{x : gen(C, \tau)\} \models_P e' : \tau' \quad \Phi \models_P C' \backslash C'_\Gamma}{\Phi(C \cup C'_\Gamma), \Gamma \models_P \textbf{let}\ x = e\ \textbf{in}\ e' : \Phi\tau'} \textsc{InfLet}$$

$$\frac{\forall \overline{\alpha}.\tau = \sigma_l}{\emptyset, \Gamma \models_P l : [\overline{\nu}/\overline{\alpha}]\tau} \textsc{InfSel}$$

$$\frac{\forall \overline{\alpha}.\tau = \sigma_k}{\emptyset, \Gamma \models_P k : [\overline{\nu}/\overline{\alpha}]\tau} \textsc{InfCon}$$

$$\frac{\begin{array}{c}\Pi_{\{l_i\}^i} = \{l_i : \forall \overline{\alpha}.\tau \to \tau'_i\}^i \\ C_i, \Gamma \models_P e_i : \rho_i \qquad \Phi \models_P \{\rho_i \leq [\overline{\nu}/\overline{\alpha}]\tau'_i\}^i\end{array}}{\Phi \bigcup_i C_i, \Gamma \models_P \{l_i = e_i\}^i : \Phi[\overline{\nu}/\overline{\alpha}]\tau} \textsc{InfStruct}$$

$$\frac{\begin{array}{c}\Sigma_{\{k_i\}^i} = \{k_i : \forall \overline{\alpha}.\tau'_i \to \tau\}^i \\ C_i, \Gamma \models_P e_i : \rho_i \qquad \Phi \models_P \{\rho_i \leq [\overline{\nu}/\overline{\alpha}]\tau'_i \to \nu\}^i\end{array}}{\Phi \bigcup_i C_i, \Gamma \models_P \{k_i \to e_i\}^i : \Phi([\overline{\nu}/\overline{\alpha}]\tau \to \nu)} \textsc{InfAlts}$$

Figure 5.7: The inference algorithm

The existence of a constraint set in the output from our inference algorithm might at first seem contradictory to our whole approach. However, these constraint sets have a very limited form, and are there just for the same purpose as the substitutions returned by Milner's algorithm W: to propagate requirements on the free variables of the assumption environment Γ [Mil78].

But instead of deciding locally on a fixed substitution that makes Γ meet its requirements, our algorithm will effectively work on cloned copies of Γ (rule InfVar), and return a constraint set that relates these clones to the original.[4] It is not until a free variable of Γ exits its scope that its constraints are collected and a satisfying substitution is computed (last premise of rules InfAbs and

[4]Here we let the symbols τ^+ and τ^- stand for the free variables of τ that occur in co- and contravariant positions, respectively.

InfLet). A key element in this step is an operation which takes a constraint set and returns only those constraints which reference variables free in Γ:

$$C_\Gamma = \{\tau \leq \rho \mid \tau \leq \rho \in C,\ fv(\tau, \rho) \cap fv(\Gamma) \neq \emptyset\}$$

We formalize the role of these generated constraint sets as follows.

Definition 5.3.6. We say that a constraint set C is a Γ-*constraint* iff $\tau \leq \rho \in C$ implies either $\tau \in fv(\Gamma)$ and $fv(\rho) \cap fv(\Gamma) = \emptyset$ or $\rho \in fv(\Gamma)$ and $fv(\tau) \cap fv(\Gamma) = \emptyset$.

Lemma 5.3.7.

1. *If C and C' are Γ-constraints then $C \cup C'$ is a Γ-constraint.*
2. *If C is a $(\Gamma \cup \Gamma')$-constraint then C_Γ is a Γ-constraint.*
3. *If C is a Γ-constraint, and $vars(\Phi) \cap fv(\Gamma) = \emptyset$, then ΦC is a Γ-constraint.*

Proof. Follows directly from the definition of Γ-constraint. □

Lemma 5.3.8. *Assume $C, \Gamma \models_P e : \tau$, $\alpha \in fv(\tau)$, and $\beta \in fv(C)$. Then:*

1. *α is a "new" variable, distinct from any variable in the context of the derivation.*
2. *either $\beta \in fv(\Gamma)$, or β is a "new" variable, distinct from any variable in the context of the derivation.*
3. *C is a Γ-constraint.*

Proof. By structural induction on e. Full details are given in appendix E.1. □

Corollary 5.3.9. *If $C, \Gamma \models_P e : \tau$ and $fv(\Gamma) = \emptyset$ then $C = \emptyset$.*

Thus, for expressions defined on the top-level of a program, the inference algorithm returns just a type if it succeeds.

Generalization plays the same role here as in Milner's W. We define this operation for the full type scheme syntax with constraints, even though we will not need this generality until section 5.5:

$$gen(C, \tau|D) = \forall \overline{\alpha}.\tau|D \quad \text{where } \overline{\alpha} = fv(\tau, D) \backslash fv(C)$$

Note that *gen* takes the Γ-constraint C as a parameter instead of Γ, since the free variables of $\tau|D$ and Γ will always be disjoint.

The vector notations in rule InfApp stand for nested application and function type construction, respectively. The possibility of letting more than one argument influence the type of an application expression can have a crucial impact on the result of the algorithm. For example, if $f : \forall\alpha.\alpha \to \alpha \to \alpha$ then $f\ cpt\ pt$ will be assigned the type *Point*, whereas $f\ cpt$ applied to pt will result in a type error (assuming pt : *Point*, cpt : *CPoint*, and *CPoint* $<_P$ *Point*). A similar argument, although seemingly less important in practice, applies to the

use of nested abstractions in rule InfAbs. Thus it is implicitly understood that rules InfApp and InfAbs are matched against as large expressions as possible.

It is worth noting that the result of the algorithm is independent of the order in which subexpressions are analyzed (it is only rule InfApp that offers any degree of freedom). A related property guarantees that function arguments can be reordered without causing any other effect than a permutation of the corresponding argument types.

Proposition 5.3.10. *Let* $f = \lambda x \to \lambda y \to e$ *and* $f' = \lambda y \to \lambda x \to e$. *Then* $C, \Gamma \models_P f : \tau \to \tau' \to \rho$ *iff* $C, \Gamma \models_P f' : \tau \to \tau' \to \rho$, *and* $C, \Gamma \models_P f\ e_1\ e_2 : \rho$ *iff* $C, \Gamma \models_P f'\ e_2\ e_1 : \rho$.

Proof. Follows from the fact that both sides of the respective implications generate the same constraint solving problems. □

Detailed examples of how the algorithm works can be found in section 5.4. We end this section with the main technical results about our inference algorithm.

Theorem 5.3.11 (Soundness). *If* $C, \Gamma \models_P e : \tau$ *then* $C, \Gamma \vdash_P e : \tau$.

Proof. By structural induction on e. Full details can be found in appendix E.2. □

Definition 5.3.12. Let $\Gamma \vdash^{HM} e : \forall\overline{\alpha}.\tau$ stand for a typing derivation in which

1. no use is made of rule TypSub,
2. all subtype constraints are empty,
3. Γ is of the form $\{x_i : \forall\overline{\alpha}_i.\tau_i\}^i$.

Such a derivation is equivalent to a derivation in the original Hindley-Milner system, or *HM* for short.

Theorem 5.3.13 (Completeness w.r.t. HM). *Assume* $\Phi\Gamma \vdash^{HM} e : \forall\overline{\alpha}.\tau$, *where* $\overline{\alpha} \notin fv(\Phi\Gamma)$. *Then* $C, \Gamma \models_P e : \rho$ *succeeds, and there is a* Ψ *such that* Ψ *unifies* C, $\Psi\Gamma = \Phi\Gamma$, *and* $\Psi\rho = \tau$.

Proof. By induction on the derivation of $\Phi\Gamma \vdash^{HM} e : \forall\overline{\alpha}.\tau$, utilizing lemma 5.3.3, and the fact that if C is a Γ-constraint and Φ *unifies* C, then $fv(\Phi C) \subseteq fv(\Phi\Gamma)$. Full details can be found in appendix E.3. □

This result gives one lower approximation of the capabilities of our inference algorithm. Another approximative characterization, which is not met by Milner's original inference algorithm, concerns completeness w.r.t. the basic type

system of Java. For the purpose of this result, we temporarily extend our calculus with type-annotated lambda abstractions, $\lambda x :: \rho \to e$, where $fv(\rho) = \emptyset$. The corresponding typing rule looks as follows:[5]

$$\frac{C, \Gamma \cup \{x : \rho\} \vdash_P e : \tau}{C, \Gamma \vdash_P \lambda x :: \rho \to e : \rho \to \tau} \text{ TypAbs'}$$

The matching clause for the inference algorithm looks almost identical, since no type variables come into play:

$$\frac{C, \Gamma \cup \{x : \rho\} \models_P e : \tau}{C, \Gamma \models_P \lambda x :: \rho \to e : \rho \to \tau} \text{ InfAbs'}$$

It should be noted here that annotated lambda-bindings are required only because we want to prove completeness, since it is trivial to show that even the identity function lacks a principal type if we abandon polymorphism as in Java. In practical O'Haskell programming, polymorphism and subtyping can be exploited simultaneously, and as our programming examples hopefully have been able to show, explicit type annotations are hardly ever necessary.

Definition 5.3.14. Let $\Gamma \vdash^J_P e : \tau$ stand for a typing derivation in which

1. there is no occurrence of any type variable,
2. all subtype constraints are empty,
3. all lambda-bindings are annotated with their type,
4. P is a partial order on *nullary* type constants where lub:s/glb:s are unique, if they exist.

We argue that derivations of this form capture the *essence* of the basic type system of Java. The correspondence is not perfect; for one thing our system lacks Java's static overloading facility. In another aspect the system $\vdash^J_P$ is stronger than Java, since it supports higher-order functions, and does not require the different branches of a datatype selection expression ($\approx$ Java's conditional operator `?`) to be trivially subtype-related. However, what we really have aimed at here is a simple formulation of a language with monomorphic subtyping and records, and Java matches this characterization closely enough to lend it its name.

Now let $C \vdash_P \Gamma \leq \Gamma'$ iff $dom(\Gamma) = dom(\Gamma')$ and $C \vdash_P \Gamma(x) \leq \Gamma'(x)$ for all $x \in dom(\Gamma)$. Our second partial completeness result is stated as follows:

Theorem 5.3.15 (Completeness w.r.t. Java). *If* $\Gamma \vdash^J_P e : \tau$ *and* $\emptyset \vdash_P \Gamma' \leq \Gamma$*, then* $\emptyset, \Gamma' \models_P e : \rho$ *succeeds,* $fv(\rho) = \emptyset$*, and* $\emptyset \vdash_P \rho \leq \tau$*.*

Proof. By induction on the derivation of $\Gamma \vdash^J_P e : \tau$. Full details can be found in appendix E.4. □

[5] Type annotated lambda-bindings can actually be expressed in the "real" calculus for annotated terms that we introduce for the purpose of *type-checking* in section 5.5 (using the equivalence $\lambda x :: \rho \to e \equiv (\lambda x \to e) :: (\rho \to \_)$. However, a formulation of our result in terms of type-checking has the drawback that it unnecessarily complicates the proof, and thus obscures the central ideas that underlie the inference algorithm.

5.4 Inference examples

In this section we will demonstrate the behaviour of our inference algorithm by means of some example runs expressed as derivations in the systems of figures 5.6 and 5.7. The type environment we assume will essentially consist of the type declarations for `Point`, `CPoint`, and `Ord` given in chapter 2 (here called *Pt*, *CPt*, and *Ord*, respectively), plus the following three type declarations:

```
struct Box a =
   fst  :: a

struct Couple a < Box a =
   snd  :: a

struct Triple < Couple CPoint =
   thd  :: Float
```

Since the translation of type declarations into a subtyping theory and other required environments can be non-trivial, we list explicitly below how the required pieces of information are extracted from the previous concrete declarations.

$$\begin{aligned}
\sigma_{fst} &= \forall\alpha.\mathit{Box}\ \alpha \to \alpha \\
\sigma_{snd} &= \forall\alpha.\mathit{Couple}\ \alpha \to \alpha \\
\sigma_{thd} &= \mathit{Triple} \to \mathit{Float}
\end{aligned}$$

$$\begin{aligned}
P_{Box} &= \emptyset \\
P_{Couple} &= \{\mathit{Couple}\ \alpha < \mathit{Box}\ \alpha\} \\
P_{Triple} &= \{\mathit{Triple} < \mathit{Couple}\ \mathit{CPt}, \mathit{Triple} < \mathit{Box}\ \mathit{CPt}\}
\end{aligned}$$

$$\begin{aligned}
\widehat{Box} &= \{\mathit{fst}\} \\
\widehat{Couple} &= \{\mathit{snd}, \mathit{fst}\} \\
\widehat{Triple} &= \{\mathit{thd}, \mathit{snd}, \mathit{fst}\}
\end{aligned}$$

$$\begin{aligned}
\Pi_{\widehat{Box}} &= \{\mathit{fst} : \forall\alpha.\mathit{Box}\ \alpha \to \alpha\} \\
\Pi_{\widehat{Couple}} &= \{\mathit{snd} : \forall\alpha.\mathit{Couple}\ \alpha \to \alpha, \mathit{fst} : \forall\alpha.\mathit{Couple}\ \alpha \to \alpha\} \\
\Pi_{\widehat{Triple}} &= \{\mathit{thd} : \mathit{Triple} \to \mathit{Float}, \mathit{snd} : \mathit{Triple} \to \mathit{CPt}, \mathit{fst} : \mathit{Triple} \to \mathit{CPt}\}
\end{aligned}$$

In the sequel we will assume that P contains the three axiom sets above plus the axiom $\mathit{CPt} <_P \mathit{Pt}$ (the relation between *Ord* and *Eq* will not be needed). The variance information relevant for our examples looks as follows:

$$\begin{aligned}
\mathit{Box}^+ &= \{1\} \\
\mathit{Box}^- &= \emptyset \\
\mathit{Ord}^+ &= \emptyset \\
\mathit{Ord}^- &= \{1\}
\end{aligned}$$

We will also assume that the assumption list Γ is such that the following equations hold:

$$\begin{array}{rcl} \Gamma(pt) & = & Pt \\ \Gamma(cpt) & = & CPt \\ \Gamma(pdict) & = & Ord\ Pt \\ \Gamma(cpdict) & = & Ord\ CPt \end{array}$$

Here follows some initial derivations that illustrate the important role of selector environments.

$$\frac{\begin{array}{c}\Pi_{\{fst,snd\}} = \{fst : \forall\alpha.Couple\ \alpha \to \alpha, snd : \forall\alpha.Couple\ \alpha \to \alpha\} \\ \emptyset, \Gamma \models_P cpdict : Ord\ CPt \qquad \emptyset, \Gamma \models_P pdict : Ord\ Pt \\ [Ord\ CPt/\nu] \models_P \{Ord\ CPt \leq \nu, Ord\ Pt \leq \nu\}\end{array}}{\emptyset, \Gamma \models_P \{fst = cpdict, snd = pdict\} : Couple\ (Ord\ CPt)}\ \textsc{InfStruct}$$

$$\frac{\begin{array}{c}\Pi_{\{fst,snd,thd\}} = \{fst, snd : Triple \to CPt, thd : Triple \to Float\} \\ \emptyset, \Gamma \models_P cpt : CPt \qquad \emptyset, \Gamma \models_P 3.14 : Float \\ [\,] \models_P \{CPt \leq CPt, Float \leq Float\}\end{array}}{\emptyset, \Gamma \models_P \{fst = cpt, snd = cpt, thd = 3.14\} : Triple}\ \textsc{InfStruct}$$

$$\frac{\begin{array}{c}\Pi_{\{fst,snd,thd\}} = \{fst, snd : Triple \to CPt, thd : Triple \to Float\} \\ \emptyset, \Gamma \models_P pdict : Ord\ Pt \qquad \emptyset, \Gamma \models_P 3.14 : Float \\ \textsc{failure} \models_P \{Ord\ Pt \leq CPt, Float \leq Float\}\end{array}}{\textsc{failure}, \Gamma \models_P \{fst = pdict, snd = pdict, thd = 3.14\} : \textsc{failure}}\ \textsc{InfStruct}$$

The first constraint solving problem above is non-trivial, so we show it in detail as well. It also illustrates most of the clauses that constitute the constraint solver. Some obvious premises are left out for space economy reasons, though, most notably $Ord = \sqcup_P\{Ord\}$ and $CPt = \sqcap_P\{CPt, Pt\}$ in the applications of CMerge, and $CPt <_P Pt$ in the CSub clause. Recall that $Ord^+ = \emptyset$ and $Ord^- = \{1\}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{[\,] \models_P \emptyset}\ \textsc{CTriv}}{[\,] \models_P \{CPt \leq CPt\}}\ \textsc{CSub}}{[\,] \models_P \{CPt \leq CPt, CPt \leq Pt\}}\ \textsc{CDepth}}{[CPt/\nu'] \models_P \{\nu' \leq CPt, \nu' \leq Pt\}}\ \textsc{CMerge}}{[CPt/\nu'] \models_P \{Ord\ CPt \leq Ord\ \nu', Ord\ Pt \leq Ord\ \nu'\}}\ \textsc{CDepth}}{[CPt/\nu'] \circ [Ord\ \nu'/\nu] \models_P \{Ord\ CPt \leq \nu, Ord\ Pt \leq \nu\}}\ \textsc{CMerge}$$

The next example is a more substantial demonstration of how the inference algorithm works. It illustrates the point of collecting constraints on the assumption environment, instead of solving them directly. The bottom inference step looks as follows:

$$\frac{\begin{array}{c}\{\nu \leq Box\ Pt, \nu \leq Triple\}, \Gamma \cup \{a : \nu\} \models_P a.fst.x + a.thd : Float \\ [Triple/\nu] \models_P \{\nu \leq Box\ Pt, \nu \leq Triple\}\end{array}}{\emptyset, \Gamma \models_P \lambda a \to a.fst.x + a.thd : Triple \to Float}\ \textsc{InfAbs}$$

Skipping the trivial application of the infix function + (of type $\mathit{Float} \to \mathit{Float} \to \mathit{Float}$), the next level of premises looks like this:

$$\frac{\begin{array}{c}\{\nu \leq \mathit{Box}\,\nu_2\}, \Gamma \cup \{a:\nu\} \models_P a.\mathit{fst} : \nu_2 \\ \emptyset, \Gamma \cup \{a:\nu\} \models_P x : \mathit{Pt} \to \mathit{Float} \\ [\mathit{Pt}/\nu_2, \mathit{Float}/\nu_4] \models_P \{\mathit{Pt} \to \mathit{Float} \leq \nu_2 \to \nu_4\}\end{array}}{\{\nu \leq \mathit{Box}\,\mathit{Pt}\}, \Gamma \cup \{a:\nu\} \models_P a.\mathit{fst}.x : \mathit{Float}}\ \textsc{InfApp}$$

$$\frac{\begin{array}{c}\{\nu \leq \nu_5\}, \Gamma \cup \{a:\nu\} \models_P a : \nu_5 \\ \emptyset, \Gamma \cup \{a:\nu\} \models_P \mathit{thd} : \mathit{Triple} \to \mathit{Float} \\ [\mathit{Triple}/\nu_5, \mathit{Float}/\nu_6] \models_P \{\mathit{Triple} \to \mathit{Float} \leq \nu_5 \to \nu_6\}\end{array}}{\{\nu \leq \mathit{Triple}\}, \Gamma \cup \{a:\nu\} \models_P a.\mathit{thd} : \mathit{Float}}\ \textsc{InfApp}$$

The first of these steps has another non-trivial premise, which looks as follows:

$$\frac{\begin{array}{c}\{\nu \leq \nu_1\}, \Gamma \cup \{a:\nu\} \models_P a : \nu_1 \\ \emptyset, \Gamma \cup \{a:\nu\} \models_P \mathit{fst} : \mathit{Box}\,\nu_2 \to \nu_2 \\ [\mathit{Box}\,\nu_2/\nu_1, \nu_2/\nu_3] \models_P \{\mathit{Box}\,\nu_2 \to \nu_2 \leq \nu_1 \to \nu_3\}\end{array}}{\{\nu \leq \mathit{Box}\,\nu_2\}, \Gamma \cup \{a:\nu\} \models_P a.\mathit{fst} : \nu_2}\ \textsc{InfApp}$$

Notice how each occurrence of the term variable a results in a new constraint on ν, and how the bounds on ν, but not ν itself, become replaced by more specific types as the inference algorithm proceeds. Had the algorithm not accumulated any constraints on ν, but instead attempted to solve these immediately, the order in which subexpressions are visited would make a difference. In our current example, the naive left-to-right order would thus result in a failure at the following subderivation:

$$\textsc{failure}, \Gamma \cup \{a : \mathit{Box}\,\mathit{Pt}\} \models_P a.\mathit{thd} : \textsc{failure}$$

On the other hand, the success of the algorithm also crucially depends on the ability to solve strictly *local* constraints as *early* as possible. To make the importance of this property appear more clearly, we provide a derived rule for conditional expressions as follows:

$$\frac{C, \Gamma \models_P e : \tau \quad C_i, \Gamma \models_P e_i : \tau_i \quad \Phi \models_P \{\tau \leq \mathit{Bool}, \tau_1 \leq \nu, \tau_2 \leq \nu\}}{\Phi(C \cup C_1 \cup C_2), \Gamma \models_P \textbf{if}\ e\ \textbf{then}\ e_1\ \textbf{else}\ e_2 : \Phi\nu}\ \textsc{InfIf}$$

This rule results if Boolean values and the conditional expression is seen as an encoding in terms of a simple datatype with two constructors that both take the unit type as arguments.[6]

$$\begin{array}{rcl}\textbf{if}\ e\ \textbf{then}\ e_1\ \textbf{else}\ e_2 & = & \{T \to (\lambda\_ \to e_1),\ F \to (\lambda\_ \to e_2)\}\ e \\ \mathit{True} & = & T\ () \\ \mathit{False} & = & F\ ()\end{array}$$

[6]The encoding would be even simpler, and more true to the real language, if we had not insisted that datatype constructors should take exactly one argument in our simple calculus.

The purpose of this encoding is to demonstrate an example where the computations of least upper bounds and greatest lower bounds interact. Consider the following inference fragment, where p is a lambda-bound variable:

$$\frac{\begin{array}{c}\{\nu \leq\ CPt\}, \Gamma \cup \{p : \nu\} \models_P p.color == Black : Bool \\ \emptyset, \Gamma \cup \{p : \nu\} \models_P pt : Pt \\ \{\nu \leq\ \nu''\}, \Gamma \cup \{p : \nu\} \models_P p : \nu'' \\ [Pt/\nu', Pt/\nu''] \models_P \{Pt \leq\ \nu', \nu'' \leq\ \nu'\}\end{array}}{\{\nu \leq\ CPt, \nu \leq\ Pt\}, \Gamma \cup \{p : \nu\} \models_P \textbf{if } p.color == Black \textbf{ then } pt \textbf{ else } p : Pt} \textsc{InfIf}$$

A least upper bound for the types of the two branches is established locally here, by a call to the constraint solver. Had this fragment instead just returned $\{\nu \leq\ CPt, \nu \leq\ \nu'', Pt \leq\ \nu', \nu'' \leq\ \nu'\}$, our simple constraint solver would be bound to fail when applied to this constraint set at a later stage, due to "short-circuiting" of the involved variable-to-variable constraints: Now, a much friendlier constraint set is returned, and the computation of a greatest lower bound for the constraints on ν can proceed successfully when p exits its scope:

$$\frac{\begin{array}{c}\{\nu \leq\ CPt, \nu \leq\ Pt\}, \Gamma \cup \{p : \nu\} \models_P \textbf{if } p.color == Black \textbf{ then } pt \textbf{ else } p : Pt \\ [CPt/\nu] \models_P \{\nu \leq\ CPt, \nu \leq\ Pt\}\end{array}}{\emptyset, \Gamma \models_P \lambda p \rightarrow \textbf{if } p.color == Black \textbf{ then } pt \textbf{ else } p : CPt \rightarrow Pt}$$

As a final example, recall that the function `min` from section 2.4 was assigned a rather limited type by our inference algorithm. Interestingly, this function can be given an alternative coding, inspired by the way overloading is handled in Haskell.

$$\textbf{let } min = \lambda d \rightarrow \lambda x \rightarrow \lambda y \rightarrow \textbf{if } d.le\ x\ y \textbf{ then } x \textbf{ else } y \textbf{ in } \ldots$$

This *min* receives a quite flexible type by the algorithm, and the generalized type scheme stored in Γ when the definition above is in scope looks as follows:

$$\Gamma(min) = \forall \alpha. Ord\ \alpha \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha$$

Instantiating this scheme in different contexts gives rise to many interesting constraint solving problems. In the following schematic algorithm run, the constraint solver has not much choice:

$$\frac{\begin{array}{c}[CPt/\nu] \models_P \{CPt \leq\ \nu, \nu \leq\ CPt\} \\ \Uparrow \\ [CPt/\nu, \nu/\nu'] \models_P \{Ord\ \nu \rightarrow \nu \rightarrow \nu \rightarrow \nu \leq\ Ord\ CPt \rightarrow CPt \rightarrow CPt \rightarrow \nu'\}\end{array}}{\emptyset, \Gamma \models_P min\ cpdict\ cpt\ cpt : CPt}$$

The example below gives more freedom, though. Here the lower bound *CPt* is chosen in favour of the upper bound *Pt*, on the grounds that it makes the resulting type more precise (see section 5.3.1 for some notes on this implicit

dependency between the constraint solver and the main algorithm).

$$\frac{\begin{array}{c}[CPt/\nu] \models_P \{CPt \leq \nu, \nu \leq Pt\} \\ \Uparrow \\ [CPt/\nu, \nu/\nu'] \models_P \{Ord\,\nu \to \nu \to \nu \to \nu \leq Ord\,Pt \to CPt \to CPt \to \nu'\}\end{array}}{\emptyset, \Gamma \models_P min\ pdict\ cpt\ cpt : Pt}$$

In the next example, the solver is again forced to choose a particular type. However, the example still shows that the flexibility of subtype inference continues to be available even when polymorphic functions are applied.

$$\frac{\begin{array}{c}[Pt/\nu] \models_P \{CPt \leq \nu, Pt \leq \nu, \nu \leq Pt\} \\ \Uparrow \\ [Pt/\nu, \nu/\nu'] \models_P \{Ord\,\nu \to \nu \to \nu \to \nu \leq Ord\,Pt \to CPt \to Pt \to \nu'\}\end{array}}{\emptyset, \Gamma \models_P min\ pdict\ cpt\ pt : Pt}$$

The effect of narrowing the context of an application can sometimes be puzzling, at least if one merely compares the types inferred with those obtainable in a fuller context.

$$\frac{\begin{array}{c}[CPt/\nu] \models_P \{CPt \leq \nu, \nu \leq Pt\} \\ \Uparrow \\ [CPt/\nu, \nu \to \nu/\nu'] \models_P \{Ord\,\nu \to \nu \to \nu \to \nu \leq Ord\,Pt \to CPt \to \nu'\}\end{array}}{\emptyset, \Gamma \models_P min\ pdict\ cpt : CPt \to CPt}$$

$$\frac{\begin{array}{c}[Pt/\nu] \models_P \{\nu \leq Pt\} \\ \Uparrow \\ [Pt/\nu, \nu \to \nu \to \nu/\nu'] \models_P \{Ord\,\nu \to \nu \to \nu \to \nu \leq Ord\,Pt \to \nu'\}\end{array}}{\emptyset, \Gamma \models_P min\ pdict : Pt \to Pt \to Pt}$$

However, as can be seen from the derivations above, the algorithm just consequently uses every piece of local information it has – if there are no lower bounds on a variable it will use the upper bounds. Considering the fact that constrained types are not inferred, the behaviour of the algorithm can mostly be understood by asking the question "on what grounds should *another* type have been inferred?". Solving the last constraint above with for example $[CPt/\nu]$ would clearly seem more far-fetched than the present pick $[Pt/\nu]$, since the type *CPt* is not even mentioned in the types of the involved expressions. The only place where the algorithm makes a truly arbitrary choice is when neither of two reasonable solutions makes the inferred type any better, as in the second but last example. Here the lower bound always has priority, as described in section 5.3.1.

In the next section we will see how this reliance on strictly local type information can be reduced somewhat, by letting the algorithm also take the *expected* type of an expression into account.

5.5 Type checking

Since we are working with an incomplete inference algorithm, the standard method of checking type annotations *after* a principal type has been inferred will not work. As a remedy, we develop an alternative approach to type annotations in this section, that makes type checking an integral part of the inference algorithm.

First we extend the expression syntax to include terms annotated with a signature

$$e ::= \ldots \mid e :: \sigma$$

and add a corresponding rule to the type system

$$\frac{C, \Gamma \vdash_P e : \sigma}{C, \Gamma \vdash_P (e :: \sigma) : \sigma} \text{ TypSig}$$

Since a signature is a type scheme, it can contain subtype constraints as well as explicit quantifiers, even though the latter are just implied in the concrete O'Haskell syntax. We will actually require here that all variables appearing in a signature be bound in the same annotation, although extending the system to deal with type variables of nested scope should present no specific problems.

The main change we introduce compared to the previous sections is that the type of an expression can now also be determined by the type expected by the context in which the expression appears. In principle such contextual information has its origin in an explicit type annotation, but we also want to make sure that expected types are propagated down to subexpressions as far as possible. For an application $e\ \overline{e_i}$ this requirement means that if the result type is expected to be τ, e should have the type $\overline{\tau_i} \to \tau$, where the τ_i are the types *inferred* for each e_i.[7]

A consequence of propagating information this way is that an expected type will generally contain a mixture of universally quantified variables (which must be protected from instantiation and in effect be treated as unique constants), and free instantiation variables originating from inferred types. This complicates the type checking algorithm slightly, but once it is handled properly, two other benefits come almost for free.

Firstly, the type checker and the inference algorithm can now be integrated, considering type inference as a special case of checking where the expected type is a unique instantiation variable. Secondly, it becomes possible to let the programmer exploit the use of *partial* type signatures, i.e. signatures where the type component may contain "holes" in the form of wildcards (_). Such

[7] Alternatively, we could let the contextual information flow in the other direction and extract an expected type for each e_i from the type inferred for e, as is done in [PT98a]. Which choice is best depends on where one expects to find the most hard-to-type lambda-abstractions, but since our algorithm is able to type most such expressions without any contextual assistance, the choice is not as crucial here as it is in [PT98a]. Our current pick is basically motivated by a slightly cleaner algorithm definition, although we are considering ways to implement a more adaptive strategy, that would try to direct the information flow to those sub-expressions that actually are lambda-abstractions.

$$\frac{\overline{\beta_i} = fv(\tau) \qquad C = \{\nu_i \leq \beta_i\}^{\beta_i \in (\tau^- \setminus \overline{\alpha})} \cup \{\beta_i \leq \nu_i\}^{\beta_i \in (\tau^+ \setminus \overline{\alpha})} \quad \Phi \models_P \{[\overline{\nu_i}/\overline{\beta_i}]\tau \leq \tau'\}}{\Phi C, \Phi[\overline{\nu_i}/\overline{\beta_i}]D, \Gamma \cup \{x : \forall\overline{\alpha}.\tau | D\} \models_P x : \Phi(\tau')} \textsc{ChkVar}$$

$$\frac{C_i, D_i, \Gamma \models_P e_i : \Phi_i(\nu_i) \qquad C, D, \Gamma \models_P e : \Phi(\overline{\Phi_i \nu_i} \to \tau)}{\bigcup_i \Phi C_i \cup C, \bigcup_i \Phi D_i \cup D, \Gamma \models_P e\ \overline{e_i} : \Phi(\tau)} \textsc{ChkApp}$$

$$\frac{\Phi \models_P \{\overline{\nu_i} \to \nu \leq \tau\} \quad C, D, \Gamma \cup \{x_i : \Phi\nu_i\}^i \models_P e : \Phi'(\Phi\nu) \qquad \Phi'' \models_P C \backslash C_\Gamma}{\Phi'' C_\Gamma, \Phi'' D, \Gamma \models_P \lambda\overline{x_i} \to e : \Phi''\Phi'\Phi(\tau)} \textsc{ChkAbs}$$

$$\frac{C, D, \Gamma \models_P e : \Phi(\nu) \qquad \sigma = gen(C, \Phi\nu | simp(D)) \quad C', D', \Gamma \cup \{x : \sigma\} \models_P e' : \Phi'(\tau) \qquad \Phi'' \models_P C' \backslash C'_\Gamma}{\Phi''(C \cup C'_\Gamma), \Phi'' D', \Gamma \models_P \textbf{let}\ x = e\ \textbf{in}\ e' : \Phi''\Phi'(\tau)} \textsc{ChkLet}$$

$$\frac{C, D', \Gamma \models_{P \oplus [\overline{\mu}/\overline{\alpha}]D} e : \Phi([\overline{\mu}/\overline{\alpha}]\tau) \qquad \Phi' \models_{P \oplus [\overline{\mu}/\overline{\alpha}]D} D' \quad \overline{\mu} \notin skol(\Phi' C) \qquad \Phi'' \models_P \{\Phi'\Phi[\overline{\nu}/\overline{\alpha}]\tau \leq \tau'\}}{\Phi''\Phi' C, \Phi''[\overline{\nu}/\overline{\alpha}]D, \Gamma \models_P (e :: \forall\overline{\alpha}.\tau | D) : \Phi''(\tau')} \textsc{ChkSig}$$

Figure 5.8: The integrated type-checking/type-inference algorithm

signatures may for example come in handy in situations where the programmer needs to specify that the type of an expression should be an application of a specific constant, but where the inference algorithm may be trusted with inferring the type arguments. We will not develop this idea any further here; we only want to emphasize that partial signatures is a natural generalization that our algorithm directly supports, provided that all wildcard types are replaced with unique instantiation variables prior to type checking.

The extended inference algorithm is shown in the form of an inference system in figure 5.8. Judgements in this system are of the form $C, D, \Gamma \models_P e : \Phi(\tau)$ and should be read "given an assumption environment Γ, a subtyping theory P, an expression e, and an expected type τ, return constraint sets C and D, and substitution Φ". The intuition behind this judgement form is captured by the soundness theorem for the extended algorithm, for which we need to introduce a few definitions.

Definition 5.5.1. We say that a type τ is a *type-pattern* if either τ is a type variable, τ is an application of a type constant other than the function arrow, or τ is $\rho' \to \rho$ such that $fv(\rho') \cap fv(\rho) = \emptyset$ and ρ is a type-pattern.

Definition 5.5.2. A triple (Γ, e, τ) is a *type checking problem* iff τ is a type-pattern, every type signature in e is closed, and the sets $fv(\Gamma)$ and $fv(\tau)$ are disjoint.

Theorem 5.5.3. *If* (Γ, e, τ) *is a type checking problem and* $C, D, \Gamma \models_P e : \Phi(\tau)$*, then* $C \cup D, \Gamma \vdash_P e : \Phi\tau$*.*

Proof. By structural induction on e, following the same general pattern as the proof of theorem 5.3.11. The condition on disjoint variables in definition 5.5.2 is used throughout to guarantee that substitutions do not affect Γ, and the more specific requirements of definition 5.5.1 are needed in the case of a lambda-abstraction, in order to enable application of the induction hypothesis. □

The rationale for returning two constraint sets is that we want to separate those constraints that restrict variables free in Γ (the Γ-constraint C) from those that directly or indirectly originate from type signatures supplied by the programmer (the set D). The latter set is a natural part of the type schemes generated in rule ChkLet. Note that by letting D contain instantiated constraints from rule ChkVar, as well as constraints that appear directly in a signature (rule ChkSig), we obtain an algorithm with the intuitive property that x and y receive the same type scheme σ in **let** $x = e :: \sigma$ **in let** $y = x$ **in** e'.

In rule ChkLet it is assumed that there exists a constraint simplification algorithm *simp* that is applied to D before a type scheme is generated. This algorithm can of course be arbitrarily sophisticated, but the simplifications described by Fuh and Mishra [FM89] seem to be sufficiently powerful for the kind of constraint simplification problems that occur in practice (recall that the constraints in D are either explicitly given in the program text, or instantiations of such constraints). Of course, *simp* is also assigned the role of flagging an error if a constraint is obviously inconsistent.

Variables which are universally quantified in an enclosing derivation are assumed to be *skolemized*, i.e. replaced with unique type constants of arity 0, whenever they appear free in a premise (see the first two premises in rule ChkSig). For this purpose we assume that the set of type constants contains a set of *skolem* constants that is sufficiently large for the program in question, and that *skol*(C) returns the set of skolem constants that occur in C. Furthermore, we apply the same notational technique as we do for indicating unique type variables, and assume that $\overline{\mu}$ stands for a vector of zero or more skolem constants that are equal only to the constants denoted by other occurrences of $\overline{\mu}$ in the same rule.

The annotated expression e in rule ChkSig is checked according to an extended subtyping theory $P \oplus [\overline{\mu}/\overline{\alpha}]D'$, where $\oplus$ denotes the operator implicitly referenced in section 5.1.2 that adds axioms to a subtyping theory and validates the generated closure. Note that all axioms in $[\overline{\mu}/\overline{\alpha}]D'$ must be monomorphic, since by assumption $fv(D') \subseteq \overline{\alpha}$. The extended subtyping theory is moreover used to solve all direct and indirect signature constraints encountered while checking e, in order to establish that they are all implied by D'. When it has

$$\frac{\tau' = inst(\sigma_l) \quad \Phi \models_P \{\tau' \leq \tau\}}{\emptyset, \emptyset, \Gamma \models_P l : \Phi(\tau)} \textsc{ChkSel}$$

$$\frac{\tau' = inst(\sigma_k) \quad \Phi \models_P \{\tau' \leq \tau\}}{\emptyset, \emptyset, \Gamma \models_P k : \Phi(\tau)} \textsc{ChkCon}$$

$$\frac{\begin{array}{c}\Pi_{\{l_i\}^i} = \{l_i : \forall\overline{\alpha}.\tau \to \tau_i'\}^i \\ \Phi \models_P \{[\overline{\nu}/\overline{\alpha}]\tau \leq \rho\} \qquad \rho_i = \Phi[\overline{\nu}/\overline{\alpha}]\tau_i' \\ C_i, D_i, \Gamma \models_P e_i : \Phi_i(\Theta_i\rho_i) \qquad \Phi' \models_P \{\Phi_i\Theta_i\rho_i \leq \rho_i\}^i\end{array}}{\Phi' \bigcup_i C_i, \Phi' \bigcup_i D_i, \Gamma \models_P \{l_i = e_i\}^i : \Phi'\Phi(\rho)} \textsc{ChkStruct}$$

$$\frac{\begin{array}{c}\Sigma_{\{k_i\}^i} = \{k_i : \forall\overline{\alpha}.\tau_i' \to \tau\}^i \\ \Phi \models_P \{[\overline{\nu}/\overline{\alpha}]\tau \to \nu' \leq \rho\} \qquad \rho_i = \Phi[\overline{\nu}/\overline{\alpha}]\tau_i \\ C_i, D_i, \Gamma \models_P e_i : \Phi_i(\Theta_i\rho_i) \qquad \Phi' \models_P \{\Phi_i\Theta_i\rho_i \leq \rho_i\}^i\end{array}}{\Phi' \bigcup_i C_i, \Phi' \bigcup_i D_i, \Gamma \models_P \{k_i \to e_i\}^i : \Phi'\Phi(\rho)} \textsc{ChkAlts}$$

Figure 5.9: Typechecking datatypes and records

been verified that no universally quantified variable escapes its scope, a fresh instance of the signature is generated, and a final check is made to ensure that the annotated type fits below the type expected by the context. Notice specifically that the rather strong demands put on D' by means of $\oplus$ actually guarantee that the extended subtyping theory is unambiguous, and hence subject reduction is bound to hold for all successfully type-checked programs.

The parts of the type checker that deal with records and datatypes are shown in figure 5.9. The rules are straightforward, with the exception of a small notational device we introduce for expressing fresh renamings: In both rules ChkStruct and ChkAlts, let each Θ_i be an invertible substitution whose domain is the set of variables it can possibly be applied to (i.e. the variables of the corresponding ρ_i), and whose range is a set of fresh variables (analogous to variables denoted by $\overline{\nu}$). The reason such renamings are needed is that we want to keep the algorithm independent of the order subexpressions are checked, even in the case where the ρ_i might have type variables in common.

Finally it should be noted that this integrated type-checking/type-inference algorithm is still incomplete, in the sense that adding a top-level type annotation is not necessarily sufficient to make a typeable program accepted by the algorithm. Seeing this is easy: the type checker relies on type inference for subexpressions in several places, and we know that the inference algorithm is incomplete.

An (almost) complete type checker could probably be developed for our system by following the ideas in [TS96b, MW97], and would be valuable for the same reason that a complete inference algorithm is. However, since the former kind of algorithm by necessity must be based on the latter, problems regarding unreadable diagnostic messages would reappear, and the programmer would experience significantly different responses from the system depending on whether a term has a type annotation or not.

Despite its less ambitious goal, our approach to type checking fulfills a vital need in conjunction with our inference strategy: it enables the programmer to guide the inference algorithm at points where it would otherwise have made a bad choice. Indeed, identifying these points on basis of error messages is likely to be facilitated by the very same properties that contribute to making the algorithm incomplete: contextual information in form of expected types is explicitly propagated top-down, and readable types are assigned not only to top-level terms, but to every subexpression as well.

Chapter 6

Reactive Objects

In this chapter we will continue the formal development of O'Haskell, by studying the dynamic semantics of reactive objects. Our treatment of this aspect of the language actually consists of two parts:

1. A translation of the full O'Haskell syntax into a form which is also valid Haskell code (the subtyping system aside), but where the reactive objects are present in the shape of a set of monadic constants.

2. An operational semantics for these constants in terms of a *reaction* relation between object/process configurations.

Both parts are needed in order to assign a meaning to the program examples we have encountered in this thesis, although one can argue that the essence of reactive objects is primarily captured in the semantics we give to the monadic constants in the second part.

As a complement to the operational semantics, we will also provide a simple "denotation" of the core language, in the form of a translation into Concurrent Haskell. Since this translation will only affect the monadic constants, the scheme actually takes the shape of an implementation of a reactive objects library in Concurrent Haskell. Moreover, to provide evidence that the two languages actually are of equal expressive power, we supply a translation in the other direction as well; i.e. an implementation of the Concurrent Haskell primitives in O'Haskell.

The chapter is organized as follows. Section 6.1 shows how the syntactic transformations into the Haskell core are carried out, while section 6.2 describes the monadic constants whose semantics will be the subject of the rest of this chapter. In section 6.3 a language of process configurations is introduced and the reaction relation is defined. Here we also show how the functional concept of expression evaluation is related to reaction. Section 6.4 gives a type soundness theorem for our semantics. In section 6.5 a *functional soundness* theorem is proved, which essentially says that reaction and evaluation are commuting relations in O'Haskell. Section 6.6, finally, contains the translations to and from Concurrent Haskell we have mentioned.

6.1 Syntactic transformations

Monads in Haskell are not defined in terms of the **do**-construct. Instead an overloaded operator `>>=` (pronounced *bind*) is provided, which together with `return` make up the signature of a monad. The type class `Monad` is actually defined as follows:

```
class Monad m where
   return :: a -> m a
   (>>=)  :: m a -> (a -> m b) -> m b
```

A derived operator `>>` is also provided, defined as

```
a >> b = a >>= \_ -> b
```

In the context of these definitions, the **do**-syntax can now be eliminated by a source-to-source translation, as follows:

do e	=	e
do e `;` $cmds$	=	e `>>` **do** $cmds$
do p `<-` e `;` $cmds$	=	e `>>= \`p `->` **do** $cmds$

O'Haskell extends this scheme with several other clauses, so that the semantics of state variables, assignments, and the **template**, **action**, and **request** constructs can be defined in terms of a source-to-source translation as well.[1] However, in contrast to the strictly local transformation above, the extended scheme must depend on non-local information in order to keep track of which state variables are in scope. We therefore informally parameterize our rules w.r.t. a *state tuple*, denoted by s, which is a tuple expression enumerating all state variables defined in the innermost enclosing **template** expression. If no template is in scope, s is assumed to equal to `()`. Since s only contains variables that are bound to be mutually distinct, s is also a valid pattern, which will be exploited below.

Assignment commands are transformed into applications of a primitive monadic constant `set`.

p `:=` e	=	`set ((\`p `->` s`)` e`)`

A static side-condition to this rule is of course that the variables in p form a subset of the variables in s.

The following rule should be applied to all expressions e referenced on the right-hand sides in the **do** elimination scheme above: If e mentions any of the variables in s, then e is transformed into

```
get >>= \s -> e
```

[1] We assume here that the extended **do**-syntax of section 2.6 has already been eliminated according to the rules of appendix A.

where `get` is a primitive constant in the `O` monad. Notice that this rule depends on a static precondition, stated in chapter 2, which prohibits shadowing of state variables.

The **template** construct locally redefines the state tuple s. Given a template expression

template p_1 `:=` e_1 `;` ... `;` p_n `:=` e_n **in** e

we let the new, local s be defined as

`(`x_1`,` ... `,`x_m`)`

where $x_1 \ldots x_m$ are the free variables of the patterns p_1 to p_n. The template expression itself is translated into

`new (\self -> (\`p_1 ... p_n `-> `s`)` e_1 ... e_n`) (\self -> `e`)`

As can be seen, the implicit variable `self` becomes explicitly bound by this step.

Translation of the method constructs is rather simple. The only thing to notice is that the connection between a method and the object it belongs to is made explicit by means of `self`.

action *cmds* = `act self (`**do** *cmds*`)`
request *cmds* = `req self (`**do** *cmds*`)`

As an example of how these translation schemes work, here follows a desugared variant of the template expression `counter` defined in chapter 2.

```
counter = new (\self -> (\val -> (val)) 0)
              (\self -> struct
                    inc = act self
                      (get >>= \(val) ->
                       set ((\val -> val) (val+1)))
                    read = req self
                      (get >>= \(val) -> return val)
```

Since all patterns used in this example are trivial, we may actually simplify the code above a bit to obtain a more legible version.

```
counter = new (\self -> 0)
              (\self -> struct
                    inc = act self
                      (get >>= \val -> set (val+1))
                    read = req self
                      (get >>= \val -> return val)
```

6.2 The naked `O` monad

After translation by the schemes defined in the previous section, O'Haskell programs come out as syntactically valid Haskell code as well, and the reactive object extension is visible only in the presence of seven primitive monadic constants. Four of these are actually the standard *state monad* operations [Wad92], here recast to our monad `O s`:

```
return :: t -> O s t
(>>=)  :: O s a -> (a -> O s b) -> O s b
set    :: s -> O s ()
get    :: O s s
```

The remaining three constants can all be seen as operations of an abstract datatype that defines the type constructor `Ref`:

```
new    :: (Ref s -> s) -> (Ref s -> t) -> Template t
act    :: Ref s -> O s () -> Action
req    :: Ref s -> O s t -> Request t
```

The intuitive semantics of these constants is as follows:

`new` s e creates an inactive object identified by a unique reference value n, initializes its state to $s\ n$, and returns $e\ n$.

`act` n e queues command e for execution by object n as soon as n is ready, and immediately returns `()`.

`req` n e queues command e for execution by object n as soon as n is ready, waits for the result of e, and returns that result.

Programming directly with these primitive constants is definitely feasible, although clearly not as convenient as using the support of the full O'Haskell syntax. It may look, though, as if this naked formulation of O'Haskell actually gives senders full freedom to assemble any (type-correct) method for an arbitrary object and have it executed on that object's behalf (essentially making objects dynamically extensible). In fact, this is also exactly the case, but *only provided* that the object in question has made its object reference available through its interface. As we have seen, an object normally exports only pre-packaged methods, where the primitive constants have already been fully applied. In such an interface, no reference value needs to be visible, and no state type is revealed.[2] Exporting the `self` reference under the type `UniRef`, as we have been discussing in section 2.5.8, is also safe, since `UniRef` does not reveal any information about the local state.

[2]This corresponds directly to what John Reynolds calls *procedural abstraction*, where information hiding is achieved by (partially) applying an access procedure to the structure that needs protection [Rey94]. Reynolds' contrasting notion, *type abstraction*, would mean exposing the object reference and its type in the interface and then encapsulating that knowledge within some scope by means of existential type quantification.

One important fact has been disregarded in the typings for `return` and `>>=` above: these names are *overloaded*, and hence the programs that result from our syntactic transformations will generally contain occurrences of `return` and `>>=` that should be typed as follows instead:

```
return :: a -> Cmd a
(>>=)  :: Cmd a -> (a -> Cmd b) -> Cmd b
```

However, this complication is purely a typing problem, since we expect the dynamic behaviour of sequencing and command injection to be identical for our two subtype-related monads. What is more, even though the primitive monadic constants benefit from the subtyping system of O'Haskell when manipulated as *values*, they will only need to be considered as elements of the "top-level" monad `O s` when *executed*.

These facts make it possible for us to consider only *normalized* command sequences in the dynamic semantics, where all occurrences of the `Cmd`-specific operations have been replaced by the corresponding (and dynamically identical) operations in the `O s` monad. To see that this is always possible to do, consider the (simplified) typing derivation

$$\frac{\dfrac{C,\Gamma \vdash_P e : \texttt{Cmd}\ \tau' \quad C,\Gamma \vdash_P e' : \tau' \to \texttt{Cmd}\ \tau}{C,\Gamma \vdash_P e\ \texttt{>>=}\ e' : \texttt{Cmd}\ \tau}}{C,\Gamma \vdash_P e\ \texttt{>>=}\ e' : \texttt{O}\ \rho\ \tau}$$

whose premises may also be used to derive

$$\frac{\dfrac{C,\Gamma \vdash_P e : \texttt{Cmd}\ \tau'}{C,\Gamma \vdash_P e : \texttt{O}\ \rho\ \tau'} \qquad \dfrac{C,\Gamma \vdash_P e' : \tau' \to \texttt{Cmd}\ \tau}{C,\Gamma \vdash_P e' : \tau' \to \texttt{O}\ \rho\ \tau}}{C,\Gamma \vdash_P e\ \texttt{>>=}\ e' : \texttt{O}\ \rho\ \tau}$$

6.3 Semantics

Having reduced the essence of reactive objects to seven primitive monadic constants, we will proceed in this section with an operational semantics for these, using a reduction-based model. The stratified semantics that is an integral part of the monadic approach will be clearly visible in the use of *two* relations between object terms, where one represents expression evaluation, and the other command execution.

In many aspects, the presentation given here borrows extensively from the definition of Concurrent Haskell given in [PGF96], for the simple reason that both languages are monadic concurrency extensions to the same core. But while the latter reference provides a model where the monadic expression `return ()` represents a terminated process, we will interpret this value as an *inactive* process/object that just maintains its state. Another fundamental difference is that whereas a Concurrent Haskell configuration consists of a mix of anonymous processes and named synchronization variables, our semantics will provide only one atomic process: the identity-carrying reactive object. It is our intention that

$$
\begin{array}{rcll}
P \parallel Q & \equiv & Q \parallel P & \\
P \parallel (P' \parallel Q) & \equiv & (P \parallel P') \parallel Q & \\
P & \equiv & Q & \text{if } P =_\alpha Q \\
\nu n.\nu m.P & \equiv & \nu m.\nu n.P & \\
\nu n.(P \parallel Q) & \equiv & P \parallel \nu n.Q & \text{if } n \notin fv(P)
\end{array}
$$

Figure 6.1: Structural congruence

the notational similarities of these two systems will enable a better comparison between the different communication models they represent.

6.3.1 Basic definitions

As a start, we will assume that the reduction relation $\mapsto$ from chapter 5 constitutes an adequate characterization of expression evaluation. Parallel tree reduction is arguably not a very precise model of the call-by-need semantics of Haskell, but it is attractively general, and illustrates the fact that the concurrency semantics of O'Haskell is not tied to any specific evaluation order. In particular, it shows that our monadic extension should be equally applicable to a call-by-value language.

However, we also need to ensure that the constants (`>>=`) and `return` combine in the usual monadic manner. Since we want to keep treating these monadic constants as primitive, this is best expressed by extending $\mapsto$ with another axiom.

$$\texttt{return}\ e\ \texttt{>>=}\ e' \quad \mapsto \quad e'\ e \quad \textsc{RedBind}$$

It is easy to verify that theorem 5.2.8 is still valid in spite of this change.

Next we go on and define a small language of process terms, that will enable us to capture the global configuration of a complete O'Haskell system.

$$
\begin{array}{rcll}
P & ::= & e^s_n & \text{Atomic process (object)} \\
 & | & P \parallel P' & \text{Parallel composition} \\
 & | & \nu n.P & \text{Reference generation}
\end{array}
$$

The atomic process e^s_n corresponds to our notion of an object referenced by n, executing a monadic expression e in the local state s. This form is restricted by the requirement that if $s : \tau$, then $n : \texttt{Ref}\ \tau$ and $e : \texttt{O}\ \tau\ \texttt{()}$. Furthermore, we require that no pair of objects are tagged with the same reference n. Section 6.4 will formalize these restrictions and show that they are all respected by the reaction rules that will be introduced below.

Following the polyadic π-calculus we also adopt the *chemical solution* metaphor, which uses a structural congruence relation $\equiv$ to abstract away from

$$\frac{P \longrightarrow P'}{P \parallel Q \longrightarrow P' \parallel Q}\ \textsc{Par}$$

$$\frac{P \longrightarrow P'}{\nu n.P \longrightarrow \nu n.P'}\ \textsc{Res}$$

$$\frac{P \equiv Q \quad Q \longrightarrow Q' \quad Q' \equiv P'}{P \longrightarrow P'}\ \textsc{Equiv}$$

Figure 6.2: Structural reaction

syntactical differences between equivalent process terms [Mil91, BB90]. Using this metaphor we may safely assume that any pair of objects in a system that are willing to interact can be brought together syntactically (as if they were molecules floating around in a chemical solution). The definition of $\equiv$ from [Mil91] is immediately applicable to our process terms; we repeat the relevant rules in figure 6.1.

6.3.2 Reaction

We are now ready to define how a solution of processes may evolve. This is essentially captured by means of a *reaction relation* $\longrightarrow$ between process terms. Thanks to the generality of the chemical framework we only have to specify the axioms of $\longrightarrow$; a non-deterministic relation between arbitrary complex process terms is automatically obtained by incorporating the structural reaction rules of the π-calculus (recapitulated in figure 6.2). We make it an implicit assumption that implementations of this non-deterministic system are *fair*; i.e. that all atomic processes which can react according to $\longrightarrow$ will also eventually do so.

Figure 6.3 shows the axioms of $\longrightarrow$. In this definition we make use of a *reaction context*, $\mathcal{M}$, in order to single out the leftmost leaf in a tree of nested (>>=) applications. $\mathcal{M}$ is defined as follows:

$$\mathcal{M} \quad ::= \quad [\,] \mid \mathcal{M} \texttt{ >>= } e$$

That is, the term identified by $\mathcal{M}$ constitutes the first command to be executed in a sequence of monadic operations. As can be seen, the role of $\mathcal{M}$ is to impose a structure on the top-level command expression of an atomic process, which is the only place where monadic commands actually get executed. The general idea of command execution and the context notation is most easily perceived in rules Set and Get, which implement the standard semantics of a state monad.

The remaining rules do not reference the state expression s of any process. Hence we take the liberty of matching against atomic processes using a simplified

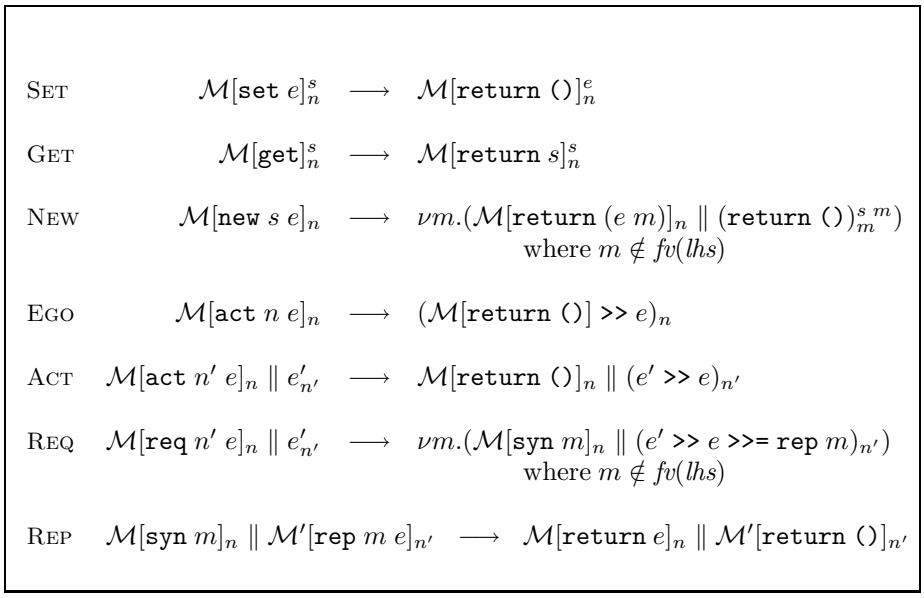

Figure 6.3: Semantics of reaction

pattern e_n; this should be interpreted as an assertion that the state component of process n is the same on both sides of a rule.

Process creation is defined by NEW. Notice how a unique reference for the new process is specified by the use of the $\nu n.P$ process form and its interaction with the congruence rules in figure 6.1. Also note that new processes are born in the inactive state represented by the command `return ()`.

Axiom EGO captures the case where the sender and the receiver of an asynchronous message are identical, while the general case is handled by ACT. Since arriving code fragments are appended to the receiver irrespective of its current activities, the semantics effectively specifies a *message buffer* for each object.

The final and most complex case is synchronous communication, which is defined in terms of two internal constants,

```
syn :: Reply a -> O s a
rep :: Reply a -> a -> O s ()
```

that are not accessible to the programmer. The type `Reply a` here stands for an infinite set of disposable tokens whose elements are only used to link a particular sender/receiver pair during the rendezvous phase of a request. In practice it is actually possible to identify such a token with the object reference of the opposite party of a request, but we prefer this slightly more circumstantial construction here since it enables a more straightforward proof of type soundness.

Rule REQ differs from ACT by the attachment of `rep` to the message sent, and

by putting the sender in a blocked state represented by $\mathcal{M}[\texttt{syn}]$. Rule Rep will subsequently be applicable to resolve this situation, provided that the receiver n' does not loop infinitely and does not cause *deadlock* by closing a chain of synchronous requests that leads back to the waiting process.

Deadlock, as well as non-termination, must thus be manually avoided if the reactivity of an O'Haskell object is going to be maintained. Notice, though, that unlike most other communication models, the existence of a deadlocked state can easily be detected in a concrete implementation of this semantics. All that is needed is that every process executing a `req` call stores the identity of the receiver in its process descriptor, after having checked that the possible chain of such links emanating from the receiver does not end at the process making the call. We have not formalized this part in order to keep the reaction axioms simple, but our current implementation supports detection and handling of deadlocks as well as other runtime exceptions (see chapter 7).

It is also important to realize that giving the programmer direct access to `syn` and `rep` (arguably together with some means of generating `Reply` identifiers) immediately would destroy the reactive character of our language. This is because free use of these primitives does not necessarily respect the invariant that they must always be introduced in matching pairs, within different processes, in order to allow progress. By only providing access to the blocking primitives through the `req` command, however, we may guarantee (under the assumptions regarding deadlock and non-termination mentioned above) that every object eventually will reach a state where it is able to handle any new message submitted to it.

6.3.3 Evaluation

So far we have not detailed how the evaluation relation $\mapsto$ is supposed to interact with $\longrightarrow$. Intuitively, an expression should be allowed to reduce by evaluation at any point without affecting the meaning of itself or any term in its context — this is actually the essence of referential transparency. In Concurrent Haskell evaluation of an atomic process is seen as a special case of reaction, and proper reaction steps can thus be interspersed with the evaluation of atomic process expressions in a non-deterministic manner.

We will follow a slightly different route, and extend the evaluation relation itself to a full-fledged relation on arbitrary process terms (see figure 6.4). This allows a process configuration to evolve in two dimensions, and the global execution history of an O'Haskell program will thus be a sequence of steps that are formed by *either* evaluation ($\mapsto$) or reaction ($\longrightarrow$).

The net effect of this formulation will still be the same as in Concurrent Haskell, namely to allow expressions to be reduced by evaluation whenever appropriate, irrespective of the current process configuration. But by keeping the two relations separate even on the level of global configurations, we are able to formulate a very simple result that captures the independence of expression evaluation from command execution. This result will be shown in section 6.5.

$$\frac{e \mapsto e' \quad s \mapsto s'}{e^s_n \mapsto e'^{s'}_n} \textsc{RedObj}$$

$$\frac{P \mapsto P' \quad Q \mapsto Q'}{P \parallel Q \mapsto P' \parallel Q'} \textsc{RedPar}$$

$$\frac{P \mapsto P'}{\nu n.P \mapsto \nu n.P'} \textsc{RedRes}$$

Figure 6.4: Reduction lifted to process terms

6.3.4 A small example

We exemplify our semantic rules by showing how the small counter program from chapter 2 reduces. A desugared variant of the program is shown below, where we have taken the liberty to introduce names for certain subexpressions in order to simplify the presentation (c.f. the equivalent translation of template `counter` that was given in section 6.1).

```
counter     = new (\self -> 0) s

s self      = struct
                  inc  = act self i
                  read = req self r

i           =    get >>= \val -> set (val+1)

r           =    get >>= \val -> return val

testCounter = counter >>= \c -> c.inc >> c.read

main env    = testCounter >>= \v -> env.putStr (show v)
```

The program environment, which is supposed to initiate execution, is modeled as a process n with an empty state `()` and an initial monadic expression `main env`, where `main` is the entry to the given program above, and `env` is an (in this example) unspecified record value that represents an interface to the computing environment. If desired, this value can be unfolded, and the interesting parts of the environment represented as distinct processes as well, but we will have no reason to do so in this simple example.

To reduce clutter, we only write out ν-binders where they are introduced, and skip trivial applications of $\mapsto$. For the same reason, some eager evaluation is also implicitly performed, and the distinction between a defined variable and

its closed right-hand side is kept on an informal level.

$$
\begin{array}{lll}
\nu n.(\texttt{main env})_n & \mapsto & \\
(\texttt{testCounter >>= \textbackslash v } \ldots)_n & \mapsto & \\
(\texttt{new (\textbackslash self -> 0) s >>= \textbackslash c } \ldots)_n & \longrightarrow & \textsc{New} \\
\nu n'.(\texttt{return (s } n\texttt{) >>= \textbackslash c } \ldots)_n \parallel (\texttt{return ()})^{\mathsf{0}}_{n'}) & \mapsto & \\
(\texttt{act } n' \texttt{ i >> req } n' \texttt{ r >>= \textbackslash v } \ldots)_n \parallel (\texttt{return ()})^{\mathsf{0}}_{n'} & \longrightarrow & \textsc{Act} \\
(\texttt{req } n' \texttt{ r >>= \textbackslash v } \ldots)_n \parallel (\texttt{return () >> i})^{\mathsf{0}}_{n'} & \mapsto & \\
(\texttt{req } n' \texttt{ r >>= \textbackslash v } \ldots)_n \parallel (\texttt{get >>= \textbackslash val -> set (val+1)})^{\mathsf{0}}_{n'} & \longrightarrow & \textsc{Get} \\
(\texttt{req } n' \texttt{ r >>= \textbackslash v } \ldots)_n \parallel (\texttt{set 1})^{\mathsf{0}}_{n'} & \longrightarrow & \textsc{Set}* \\
(\texttt{req } n' \texttt{ r >>= \textbackslash v } \ldots)_n \parallel (\texttt{return ()})^{\mathsf{1}}_{n'} & \longrightarrow & \textsc{Req}* \\
\nu m.((\texttt{syn } m \texttt{ >>= \textbackslash v } \ldots)_n \parallel (\texttt{return () >> r >>= rep } m)^{\mathsf{1}}_{n'}) & \mapsto & \\
(\texttt{syn } m \texttt{ >>= \textbackslash v } \ldots)_n \parallel (\texttt{get >>= \textbackslash val -> return val } \ldots)^{\mathsf{1}}_{n'} & \longrightarrow & \textsc{Get} \\
(\texttt{syn } m \texttt{ >>= \textbackslash v } \ldots)_n \parallel (\texttt{return 1 >>= rep } m)^{\mathsf{1}}_{n'} & \mapsto & \\
(\texttt{syn } m \texttt{ >>= \textbackslash v } \ldots)_n \parallel (\texttt{rep } m\ \texttt{1})^{\mathsf{1}}_{n'} & \longrightarrow & \textsc{Rep} \\
(\texttt{return 1 >>= \textbackslash v -> env.putStr (show v)})_n \parallel (\texttt{return ()})^{\mathsf{1}}_{n'} & \mapsto & \\
(\texttt{env.putStr "1"})_n \parallel (\texttt{return ()})^{\mathsf{1}}_{n'} & &
\end{array}
$$

To illustrate the non-determinism inherent in the reaction relation we give an alternative reduction sequence for steps marked with $*$ above, which also shows buffering of a message to an active object.

$$
\begin{array}{lll}
\vdots & \longrightarrow & \textsc{Get} \\
(\texttt{req } n' \texttt{ r >>= \textbackslash v } \ldots)_n \parallel (\texttt{set 1})^{\mathsf{0}}_{n'} & \longrightarrow & \textsc{Req}* \\
\nu m.((\texttt{syn } m \texttt{ >>= \textbackslash v -> } \ldots)_n \parallel (\texttt{set 1 >> r >>= rep } m)^{\mathsf{0}}_{n'}) & \longrightarrow & \textsc{Set}* \\
(\texttt{syn } m \texttt{ >>= \textbackslash v } \ldots)_n \parallel (\texttt{return () >> r >>= rep } m)^{\mathsf{1}}_{n'} & \mapsto & \\
\vdots & &
\end{array}
$$

6.4 Type soundness

In this section we will state and prove the basic type soundness results about our semantics. For this purpose we will introduce the concept of *well-typed* process terms, even though such terms are only part of the execution model of the language and hence not subject to typing errors in the ordinary sense.

$$\frac{\Gamma \vdash e : \texttt{O}\ \rho\ \texttt{()} \quad \Gamma \vdash s : \rho \quad \Gamma \vdash n : \texttt{Ref}\ \rho}{\Gamma \vdash e^s_n \text{ well-typed}}\ \textsc{TypObj}$$

$$\frac{\Gamma \vdash P \text{ well-typed} \quad \Gamma \vdash Q \text{ well-typed}}{\Gamma \vdash P \parallel Q \text{ well-typed}}\ \textsc{TypPar}$$

$$\frac{\Gamma \cup \{n : \tau\} \vdash P \text{ well-typed}}{\Gamma \vdash \nu n.P \text{ well-typed}}\ \textsc{TypRes}$$

The first of these rules formalizes the type restrictions on atomic process terms that were already stated in section 6.3.1, while the others just constitute obvious generalizations. The main results we will show is that typing judgements formed by these rules are respected by both the evaluation and reaction relations on process configurations.

The first part in this undertaking is trivial thanks to theorem 5.2.8, and we simply state the indicated result without proof.

Theorem 6.4.1. *If* $\Gamma \vdash P$ well-typed *and* $P \mapsto Q$*, then* $\Gamma \vdash Q$ well-typed*.*

The second part concerns a relation formulated in terms of contexts, and hence we need to introduce typing rules for these as well. Essentially, a context is typed as if it were a function from the type of the "hole" to the type of the result.

$$\frac{}{\Gamma \vdash [\,] : \mathtt{O}\ \rho\ \tau \to \mathtt{O}\ \rho\ \tau}\ \textsc{TypMTriv}$$

$$\frac{\Gamma \vdash \mathcal{M} : \mathtt{O}\ \rho\ \tau \to \mathtt{O}\ \rho\ \tau' \quad \Gamma \vdash e : \tau' \to \mathtt{O}\ \rho\ \tau''}{\Gamma \vdash \mathcal{M}\ \texttt{>>=}\ e : \mathtt{O}\ \rho\ \tau \to \mathtt{O}\ \rho\ \tau''}\ \textsc{TypMBind}$$

The following lemmas show that typing an expression in terms of a context is equivalent to inferring its type directly on basis of its structure.

Lemma 6.4.2. *If* $\Gamma \vdash \mathcal{M} : \mathtt{O}\ \rho\ \tau' \to \mathtt{O}\ \rho\ \tau$ *and* $\Gamma \vdash e : \mathtt{O}\ \rho\ \tau'$*, then* $\Gamma \vdash \mathcal{M}[e] : \mathtt{O}\ \rho\ \tau$*.*

Lemma 6.4.3. *If* $\Gamma \vdash \mathcal{M}[e] : \mathtt{O}\ \rho\ \tau$ *then there is a* τ' *such that* $\Gamma \vdash \mathcal{M} : \mathtt{O}\ \rho\ \tau' \to \mathtt{O}\ \rho\ \tau$ *and* $\Gamma \vdash e : \mathtt{O}\ \rho\ \tau'$*.*

We also need to show that typing judgements are closed under structural congruence.

Lemma 6.4.4. *If* $\Gamma \vdash P$ well-typed *and* $P \equiv Q$*, then* $\Gamma \vdash Q$ well-typed*.*

Proof. By simple induction on the derivation of $P \equiv Q$. □

We are now ready to prove the remaining type soundness theorem, which says that that even the reaction relation preserves typing judgements.

Theorem 6.4.5. *If* $\Gamma \vdash P$ well-typed *and* $P \longrightarrow Q$*, then* $\Gamma \vdash Q$ well-typed*.*

Proof. By induction on the derivation of $P \longrightarrow Q$. □

6.5 Functional soundness

We have not developed any theory around our semantics so far, although this would certainly be an interesting research project. The similarities between our formulation and the monadic semantics of Concurrent Haskell are so strong, though, that we expect the main results about the latter language to directly

carry over to our case. This includes the intuitive property that denotational equivalence of terms implies testing equivalence [PGF96].

However, we believe that the basic intuition behind the stratified semantics of a monadic approach is more accurately captured by a result which states that the deterministic, purely functional semantics of expression evaluation is not affected by its inclusion into an imperative and reactive context. So what we will show, as our main *functional soundness result*, is that the notions of reaction and evaluation are effectively independent. This means that whenever a process configuration permits transitions according to $\longrightarrow$ as well as $\mapsto$, it does not matter which path is taken, since any transition not performed in the first round can always be reconsidered at a later stage. In other words, we will show that the relations $\longrightarrow$ and $\mapsto$ commute.

As an intermediate step towards this result, we need to lift the concept of reduction to the level of reaction contexts:

$$\frac{}{[\,] \;\mapsto\; [\,]}\ \textsc{RedMTriv}$$

$$\frac{\mathcal{M} \;\mapsto\; \mathcal{M}' \quad e \;\mapsto\; e'}{\mathcal{M} \texttt{ >>= } e \;\mapsto\; \mathcal{M}' \texttt{ >>= } e'}\ \textsc{RedMBind}$$

The following lemmas state that reduction using contexts is just a shorthand for ordinary reduction.

Lemma 6.5.1. *If $\mathcal{M} \mapsto \mathcal{M}'$ and $e \mapsto e'$, then $\mathcal{M}[e] \mapsto \mathcal{M}'[e']$.*

Lemma 6.5.2. *If $\mathcal{M}[e] \mapsto e''$ and $e \neq$ `return _`, then there is an $\mathcal{M}'$ and an e' such that $e'' = \mathcal{M}'[e']$, $\mathcal{M} \mapsto \mathcal{M}'$, and $e \mapsto e'$.*

Note that if e actually is an application of `return` in the second lemma, axiom RedBind might be applicable and an $\mathcal{M}'$ with the same structure as $\mathcal{M}$ need not exist. It can easily be verified, though, that none of the rules in figure 6.3 match against a monadic expression where this is the case, which is sufficient for our purpose.

We will also need the following two lemmas:

Lemma 6.5.3. *If $x \in fv(e)$ and $e \mapsto e'$, then $x \in fv(e')$.*

Proof. Immediate from the standard assumption that substitution implicitly performs α-conversion in order not to capture free variables. □

Lemma 6.5.4. *If $P \equiv Q$ and $P \mapsto P'$, then there is a Q' such that $P' \equiv Q'$ and $Q \mapsto Q'$.*

Proof. By simple induction on the derivation of $P \equiv Q$. □

The main theorem can now be stated and proved as follows:

Theorem 6.5.5 (Functional soundness). *If $P \longrightarrow Q$ and $P \mapsto P'$, then there is a Q' such that $P' \longrightarrow Q'$ and $Q \mapsto Q'$.*

Proof. By induction on the derivation of $P \longrightarrow Q$. □

6.6 O'Haskell vs. Concurrent Haskell

In this section we will illustrate how the O'Haskell core relates to Concurrent Haskell, by means of translations between the two languages in both directions. To the Concurrent Haskellate reader this can be seen as a form of denotational semantics for the O'Haskell primitives; however, the translation exercise is probably more relevant as an evidence that the two languages can implement each other, and thus have the same expressive power.

We begin with an implementation of O'Haskell in terms of the Concurrent Haskell primitives; the opposite translation follows in subsection 6.6.2.

6.6.1 Implementing O'Haskell in Concurrent Haskell

The interpretation of the primary O'Haskell type `O`, the type of reactive commands, is built on top of (Concurrent) Haskell's `IO` monad, where the commands are given access to a mutable variable that holds the local state of the currently executing object.

```
type O s t = IORef s -> IO t
```

The type `O` is recognized as the standard reader monad.[3]

```
returnO  a  = \_ -> return  a
e `bindO` f = \v -> do a <- e v
                       f a v
```

Implementing the state-related operations boils down to accessing and manipulating the mutable variable supplied to the commands.

```
get   = \v -> readIORef v
set s = \v -> writeIORef v s
```

The type of object references encapsulates a message queue (called *channel* in [PGF96]) of commands to be executed by the object.

```
type Ref s = Chan (O s ())
```

Sending an asynchronous action to an object implies writing the corresponding command to this object's channel.

```
act r c = \_ -> putChan r c
```

The synchronous request is a little more involved, here we create a temporary `MVar` to mediate the answer.

```
req r c = \_ -> do ans <- newMVar
                   putChan r
                           (\v -> do a <- c v
                                     putMVar ans a)
                   takeMVar  ans
```

[3] To avoid any confusion regarding overloading, we let the basic monad operations of O'Haskell be denoted by `returnO` and `bindO` in this subsection.

Each object has an associated server thread which forever reads commands from the object channel and executes them.

```
objproc    :: IOVar s -> Chan (O s ()) -> IO  ()
objproc v r = do c <- getChan r
                 c v
                 objproc v r
```

The last primitive to define is `new`, which creates a fresh mutable variable for the state, and a new command queue. Finally, `new` forks off a server thread for the new object and returns the applied interface.

```
new s iface = \_ -> do r <- newChan
                       v <- newIORef (s r)
                       forkIO (objproc v r)
                       return (iface r)
```

6.6.2 Concurrent Haskell in O'Haskell

We consider the following operations in Concurrent Haskell's `IO` monad:

```
returnIO :: a -> IO  a
bindIO   :: IO a -> (a -> IO b) -> IO b
forkIO   :: IO () -> IO ()
newMVar  :: IO (MVar a)
putMVar  :: MVar a -> a -> IO ()
takeMVar :: MVar a -> IO a
```

Our implementation of the `IO` monad is defined as a composition of a continuation monad and a reader monad.

```
type IO a = Ref () -> (a -> O () ()) -> O () ()

returnIO a   = \_ c -> c a
e `bindIO` f = \s c -> e s (\a -> f a s c)
```

The `Ref` component in the `IO` type reflects the fact that *any* expression of type `IO t` might involve indefinite blocking, so we must be able to produce callbacks, i.e. action values, for the currently executing process on the fly.

Forking off a process is interpreted as the creation of a stateless object exporting a single action-valued interface, which is immediately triggered. Note the use of the implicitly bound variable `self`.

```
forkIO p = \_ c -> do o <- t; o; c ()
 where t = template  in action p self return
```

An `MVar` becomes a special kind of object, with the following interface:

```
struct MVar a =
   put  :: a -> Request ()
   take :: (a -> Action) -> Action
```

The intention is that `put` updates the state of its `MVar` with a new item, while `take` announces the readiness of some process to consume a possible item stored inside the `MVar` (c.f. section 3.5). A **request**, rather than an **action**, is necessary in the type of `put` in order to mimic the error-handling semantics of Concurrent Haskell.

Assuming there is a template `mvar` for creating `MVar` objects, the implementation of `newMVar` and `putMVar` is straightforward:

```
newMVar     = \_ c -> do v <- mvar; c v
putMVar v a = \_ c -> do v.put a; c ()
```

Since executing `takeMVar` marks the end of the currently executing process fragment, the translation does not call the given continuation, but wraps it up in a fresh callback using the `Ref` identity of the currently executing process. This new action is sent to the `MVar` in question, as the calling process returns to a resting state.

```
takeMVar v = \s c -> v.take (\a -> act s (c a))
```

Finally we give the code for `mvar`. It is a variant of the queue encoding in section 3.5 that limits the number of stored items to at most one. Types for the state variables are included as a reading aid, but we have to put these inside comments since we do not (yet) support scoped type variables in O'Haskell.

```
mvar :: Template (MVar a)
mvar =
   template
      val    := Nothing    -- :: Maybe a
      takers := []         -- :: [a -> Action]
   in struct
      put a = request
         case takers of
            []   -> case val of
                       Nothing -> val := Just a
                       Just _  -> error "putMVar"
            t:ts -> takers := ts
                    t a

      take t = action
         case val of
            Nothing -> takers := takers ++ [t]
            Just a  -> val := Nothing
                       t a
```

Judging from the sheer amount of code required, the translation of Concurrent Haskell into O'Haskell looks slightly more complex than the converse encoding. It should be borne in mind, though, that the interpretation in section 6.6.1 also involves an implementation of the abstract data type `Chan`, which roughly

corresponds to the encoding of MVars above. We therefore conclude that the two languages are equal in expressive power, and that neither language can be coined more 'primitive' or 'basic' than the other.

Chapter 7

Implementation

In this chapter we will briefly discuss our concrete O'Haskell implementation, and show that neither the subtyping system nor the monad of reactive objects presupposes any elaborate implementation techniques at either compile-time or run-time. Our present implementation is a modified and extended variant of the popular interactive Haskell system *Hugs* [Jon96a], and in accordance with our previously used naming scheme, we call our extended system *O'Hugs*. The current version of O'Hugs is downloadable from the author's homepage (`www.cs.chalmers.se/~nordland`), although it must be noted that this release is still to be considered as beta, and the best documentation for it so far is actually the present thesis.

The chapter is organized as follows. In section 7.1 we give an overview of the major modifications we have carried out to the interactive system itself. Runtime representations and code generation issues for our extensions are discussed in section 7.2, while section 7.3 outlines how the current techniques can be improved in a future optimizing O'Haskell compiler. Then in section 7.4 we will make some notes on how the subtype inference algorithm of chapter 5 is actually implemented in the full version of the language.

7.1 O'Hugs

Hugs is an interactive system that processes program scripts by compiling them into a compact, machine-independent assembly format, and this description holds for O'Hugs as well. The compilation process consists of lexing and parsing, a static analysis pass which among other things performs dependency analysis on definitions, type-checking/type-inference with resolving of overloading, various syntactic transformations, a lambda-lifting pass which turns a program into a set of top-level *super-combinators*, and then finally code-generation for these in the form of instructions for an abstract graph-reduction machine. Loaded with a set of compiled program scripts, the interactive system can then either be used as a simple expression evaluator, or it can be instructed to execute a *procedure*,

provided that the expression denoting it has the expected monadic type. In our case this means an expression of type *Env* `-> Cmd ()`, where *Env* can be the type of any of the supported computing environments (see section 2.5.11). There is also a stand-alone variant of the system which silently loads a given set of scripts and then automatically invokes the procedure `main`.

The inner workings of Hugs (which is implemented in C) is covered in detail in an excellent report written by Hugs' originator Mark Jones [Jon94]. Although the report actually discusses Gofer, a predecessor of Hugs, the differences are negligible when put in relation to the vast amount of valid and useful information that is provided. We will not repeat anything of that material here; instead we will just outline the steps we have taken in order to turn Hugs into an implementation of O'Haskell. Moreover, most of these steps are actually quite obvious (e.g. extending the static analysis with cases for our new forms of expressions and type declarations, or implementing the syntactic transformations described in the previous chapter), so we will concentrate on the few parts where our modifications have had to be a bit more involved. These parts solely concern the implementation of subtype inference.

7.1.1 Top-down type-checking

The original Hugs type-checker is a C function which takes a pointer to the syntax tree of the expression to be checked, and returns a pointer to an equivalent expression where overloading has been made explicit. The inferred type, as well as any type-class constraints that might result from the use of overloaded identifiers, are all returned via global variables. Type annotations are checked *after* a principal type for the annotated expression has been inferred, which appears to be the common method in all current implementations of Haskell.

In order to let type annotations actively guide our incomplete inference algorithm where it would otherwise fail, we have been forced to restructure the original type-checker along the ideas of section 5.5. So in our variant, the inferred type for an expression is no longer returned, but only constructed internally within a call to the type-checker for matching against an *expected* type taken as an extra argument. This means that calls to the unification algorithm (i.e. our constraint solver) appear more frequently in our modified code, but this is actually to the benefit of the system user since the type error messages can consequently be made more informative. Moreover, programmer-supplied type information is now propagated top-down, so if an incomprehensible type error is found, temporarily inserted type annotations can play a much more active role in locating the source of the error than what is usually the case. We believe that switching to top-down type-checking would be valuable for Hugs as well, even if no attempt is made to implement subtyping.

7.1.2 Solving constraints

Replacing unification with constraint-solving must of course imply some serious changes to the original code, yet not to the great extent that one perhaps would

expect. In fact, our constraint solver still continues to use most of the code that constitutes the original unification algorithm, with the notable addition of a branch that implements the CSub clause of our specification (see figure 5.6). The trick that makes this possible is that we employ a subsequent algorithm that handles the CMerge steps, so we just need to record the existence of a new upper or lower bound for a variable when a variable-to-type-constructor constraint is encountered during the first constraint-solving pass. Such bounds can moreover be recorded in the existing structure that holds the status of type variables, so instead of setting a tag which indicates that variable `v` is *unified* with type expression `t`, we just use another tag saying that `v` lies *somewhere in between* the bounds given by a list of polarized type expressions `t`. The only extra book-keeping needed here is the updating of a global list which collects all variables that currently have accumulated bounds.

Our second algorithm finds its way through this list on basis of *usage counts* for each variable encountered during the first pass. This idea is also borrowed from the unification algorithm developed by Martelli and Montanari [MM82], who obtain their efficiency result by solving equality constraints in order of dependency. Our second pass behaves basically in the same way: it tries to find a variable with a usage count of zero, then it tries to solve the constraints accumulated for this variable (according to clause CMerge); if this succeeds, it updates the appropriate usage-counters and calls the first algorithm on the sub-constraints given by CMerge, before it starts all over again, until there are no more variables with accumulated bounds. Should a variable with a zero usage count not exist, we know that there must be a cycle in the constraint dependency graph, but our current strategy is to continue and let this error be trapped when we have a *directly recursive* constraint, for the sake of a more comprehensible error message.[1]

The "cloning" of type variables prescribed by the type-checker rule ChkVar is also implemented by a dedicated tag pointing out a list of bounds in the status-table for type-variables. When the type-checker leaves a scope, the cloned variables reached from the detached part of the environment are turned into variables with accumulated bounds, and the second pass of the constraint-solving algorithm above is called upon to find a solution. The main difference between cloned variables and variables with accumulated bounds is that cloned variables can have bounds that are themselves variables; hence some variable-to-variable constraints might need to be solved when the conversion above is made. The cloning mechanism is also used to implement the fresh renamings prescribed by rules ChkStruct and ChkAlts.

It should be noted that none of these modifications alter the efficiency of the constraint-solver — or, for that matter, the main type-checking algorithm — in any essential way. In particular, the addition of an extra constraint-solving pass is already a key component of a very efficient implementation of unification.

[1] We cannot resist mentioning an amusing bug this strategy once gave rise to: an erroneous program was caught by the type-checker, but the type-checker failed to terminate while pumping out an undeniably illustrative error message: `Constraints are recursive`, `Constraints are recursive`, `Constraints are recursive`, ... !

Our practical experience with the algorithm also seems to verify this — type-checking in O'Hugs is only marginally slower than in Hugs, even for large Haskell programs. This result has furthermore been obtained without any profiling or fine-tuning of the implementation whatsoever.

7.2 Program execution

When static analysis has been performed, types have been checked or inferred, and the syntactic enhancements have been transformed away, we are left with the questions of how to represent records and datatypes so that subtyping becomes feasible, and how to implement the constants that constitute the `O` monad. We will deal with each of these issues in turn.

7.2.1 Subtyping at run-time

Hugs uses a quite simplistic implementation technique for datatypes and case selection: each term constructor in the whole program is represented by a unique token, and case expressions are just translated into series of equality tests on such tokens. This makes our subtyping extension for datatypes extremely easy to implement, since every case expression is already prepared to scrutinize values belonging to any conceivable subset of its listed constructors. The only effect subtyping has on the existing datatype implementation is that the question of whether a pattern is *failure-free* now ultimately becomes dependent on at which type a pattern is considered. We have avoided the complications regarding this matter altogether, and defined all datatype constructor patterns as *not* being failure-free; a very minor change to the semantics of Haskell, which moreover can be undone by the insertion of *irrefutable* pattern markers (see the current Haskell report for more details [Pet97]).

One obvious idea is to implement records along the same line; i.e. to compile record selection into code that searches a table of tagged expressions for a specific tag. But it turns out that this effect can actually be achieved with much less effort: by turning records into case expressions, and selector applications into applications in the reverse order, we get the dynamic behaviour we want (including correct record subtyping) absolutely for free! Our syntactic transformations are thus extended with the following rules:

$$\begin{aligned}&\textbf{struct}\ l_1 = e_1\ ;\ \ldots\ ;\ l_n = e_n\\ &\quad = \backslash x\ \texttt{->}\ \textbf{case}\ x\ \textbf{of}\ l_1\ \texttt{->}\ e_1\ ;\ \ldots\ ;\ l_n\ \texttt{->}\ e_n\end{aligned}$$

$$e.l\ =\ e\ l$$

As a consequence of this scheme, the symbol-table information for a selector needs to be stored in such a way that the selector can automatically take on the role of a constructor during code generation. It should also be noted that the rules above are *not* type-preserving (note especially the discrepancy between the second rule and the treatment of record selection in section 5.1.2). However, this

liberty is acceptable here since the transformation is dynamically correct, and we are moreover only applying it to programs that have already been accepted by the type-checker in a previous pass.

7.2.2 Implementing the `O` monad

To implement the standard Haskell IO monad on top of its ordinary graph-reduction machinery, Hugs uses a continuation passing technique which supplies each monadic operation with two continuations; one for the normal thread of control, and one for exceptions. The difference between evaluating a command expression and actually executing it then just amounts to whether one supplies the appropriate continuation arguments or not. We follow the same route for our `O` monad, and we are actually able to directly reuse the code that implements normal command sequencing, exception handling, and basic file operations (although the types we assign to those primitives are of course different).

An object/process is represented by a dynamically allocated node with three components in our system. Two of these are directly suggested by the semantics in chapter 6: a pointer to the current state, and a list of method bodies that have yet to be executed by the object. Component number three is the temporary *request-pointer* mentioned in section 6.3.2 that facilitates deadlock detection.

Objects with non-empty command-lists and *NULL*-valued request-pointers are furthermore linked together in a *ready-list* consumed by the *scheduler*, which is the run-time system routine that gets control when a procedure-valued expression is entered on the O'Hugs command line. The scheduling strategy is currently very simple; objects are switched in a round-robin manner whenever a method terminates, but no preemptive scheduling is implemented. Neither have we made any attempts so far to make scheduling decisions on basis of the load balance in the system, although the scheduler prioritizes ongoing work in the sense that it only passes along external input when no processes are active in the ready-list.

The communication primitives are implemented as could be expected. In both cases the embedded command is appended to the command-list of the selected object, which is then added to the ready-list if it is not already there. Two factors are unique to the implementation of the `req` primitive. Firstly, the deadlock-check is performed, as outlined in section 6.3.2. Secondly, two dynamic continuations are created; one that sees to that the client object is inserted into the ready-list when the request terminates normally, and one that triggers a `RequestAborted` exception in the client if the server object encounters an uncaught exception during the handling of a request.

Implementation of the remaining primitives in the `O` monad is entirely straightforward.

7.3 Possible improvements

The simplistic implementation techniques presented above are only justifiable in an interactive setting where compilation speed is valued higher than execution efficiency. Still we think that the current implementation at least shows that O'Haskell does not *require* any sophisticated techniques or analyses in order to be practically implementable. However, to back up our claim that O'Haskell can be *efficiently* realized, we will outline some more refined solutions to the implementation problems above in this section.

In the context of these points, though, we would like to make it clear that we consider the state-of-art in functional language implementations to be the starting-point for any discussion concerning the efficiency of O'Haskell in general. After all, efficient dynamic storage, unboxed data representations, function inlining, etc, are aspects vital to the performance of *any* functional language, irrespective of whether we consider any O'Haskell-specific features or not. For a broad introduction to how these general problems are tackled in modern functional languages, see e.g. [PJ96] or [App92].

Concentrating on the extensions we have added to Haskell, we identify three areas that are vital to performance.

7.3.1 Record layout

An efficient implementation will of course need to store record values as contiguous blocks of memory, where selection is equivalent to indexing with a constant value for each selector. A similar technique is common for datatypes, where selection is done by indexing with a constructor-dependent offset in a so called *jump-table*. Single-rooted subtyping does not affect this scheme in any way, but once we allow type extension with more than one base-type, a problem arises as to what fixed offset to assign to each constructor/selector.

However, this problem is certainly not new to O'Haskell, as every language that supports some form of multiple inheritance must deal with it somehow. The classical solution for C++ should be directly applicable to our case [Str89], although we believe that a slightly simplified variant ought to be sufficient. In this variant, the record type

```
struct A < B,C =
   x,y,z :: D
```

is automatically transformed into

```
struct A < B =
   fromAtoC :: C
   x,y,z    :: D
```

and wherever the type-checker finds that a value `e` of type `A` is used as a value of type `C`, `e` is replaced with `e.fromAtoC`. The `C` component is here assumed to be stored in its own memory block, keeping its ordinary selector-to-offset assignments. This technique is actually quite similar to the suggested implementation

of sub-classing in Haskell [HHJW94], but with the important distinction that our hierarchies are automatically flattened along their leftmost dependency thread (i.e. the components of `B` are still to be found in the same memory block as those for `A`, silently inserted before the pointer corresponding to `fromAtoC`).

Depth subtyping causes a slight complication to this scheme, though, since subsumption might now occasionally have an operational meaning. That is, if we in some context need to interpret (say) a list `e` containing values of type `A` as a list of `C` elements, each element in `e` must be uniformly coerced; i.e. we must replace `e` with `map (.fromAtoC) e` as part of the type-checking process. Furthermore, since depth subtyping is not restricted to just lists, the compiler must be able to generate map-like functions for every parameterized type that is either co- or contravariant in some argument. Note, though, that this complication only arises in the presumably infrequent cases where depth subtyping is used in combination with the non-standard coercions outlined above. For example, our list expression of type `[A]` can readily be interpreted as a value of type `[B]`, since a subsumption step from `A` to `B` has no operational consequences in our proposed scheme.

7.3.2 Method invocation

The implementation of method invocation described in the previous section might appear a bit costly, since new commands must be appended to a list that might be arbitrary long. However, linked imperative queue structures with constant times for insertion as well as removal are easy to implement, and we take such an approach for granted in a compiled implementation of the language. The normal and exceptional continuations pushed on the execution stack whenever an object is executing code can probably also be utilized to reduce the constant cost of these operations. Furthermore, in a system that supports preemptive scheduling, only the queue operations need to be protected from concurrent access, which means that a synchronization mechanism based on a spin-lock for each object should be sufficient.

The current scheme for translating the method syntax into monad operations prescribes that a method is represented as a closure at run-time, containing a code-pointer and a reference to the actual object on which to operate (i.e. `self`). This means that not only will every method of every object occupy a slot in the interfaces exported, there will also be a space overhead that comes from the fact that each method stores its own copy of `self` in its closure. Should it turn out that this space consumption is a significant problem in practice, there is always the option of resorting to the standard *self-application* technique for method invocation whenever a method occurs in an interface that is a subtype of the predefined type `Object` (see [AC96] as well as the definition of `Object` in section 2.5.8).

7.3.3 State updates

The semantics for assignments given by the syntactic translations in section 6.1 is clearly inefficient — for every assignment to a state variable, the whole local state-tuple is rebuilt. However, a condition that enables state variables to be updated in place is easy to establish: just make the primitive state-reading operation `get` inaccessible by the programmer! If there is no way of obtaining a name for the complete current state, there are no expressions whose semantics may be compromised by making a destructive state assignment. Ordinary references to state variables must now of course be implemented by extraction from the state-tuple whenever a state-dependent monadic expression is built, but that is just the natural technique one would expect from any imperative implementation of encapsulated objects.

The array-update syntax described in section 2.6 can be efficiently implemented virtually along the same lines. Two complications may arise here, though. Firstly, the programmer now has a name on the structure that needs to be destructively updated, so conservative copying might be necessary at those places where this name is used for another purpose than indexing, as we have mentioned. Furthermore, the lazy semantics of Haskell causes some extra trouble here, since array indexing can only be implemented as an offset operation if it is known that the index expression can safely be evaluated without changing the termination properties of the program. We conjecture that in most realistic cases this so called *strictness* property should be very easy to establish, but where this is not the case, copying must again be employed as a conservative precaution.

7.4 Subtyping in the full language

7.4.1 Overloading

The overloading system of Haskell has not interfered in any essential way with our implementation of subtype inference [HHJW94]. We explain this in part by our general strategy for solving constraints: since we are not interested in complete type inference we can choose to ignore all type-class constraints while subtype constraints are being solved. A solution to the latter kind of constraints moreover bears no trace of the subtyping extension; in particular, the context reduction algorithm that either accepts or rejects class constraints still sees only isolated type constants — it is not aware of any relation that makes type constants overlap.

Here is one area where one could imagine some sort of interaction between the subtyping and overloading systems. In many cases it might be that, for example, one only wants to define equality for the top type of some subtype hierarchy. So if the context reduction algorithm then finds a class constraint `Eq A` which has no matching instance declaration, *but* there exists an `Eq` instance for some supertype `B` of `A`, it would make sense to replace `Eq A` with `Eq B` and

then accept the constraint (provided that `B` is actually the least supertype of `A` that is an instance of `Eq`). We are currently considering such an extension.

Otherwise, the only place where subtyping currently affects the overloading system is in the implementation of derived instances for datatypes formed by extension. We have not found any good technique for implementing orderings and the like for such types yet (much due to the same problems that are discussed in subsection 7.3.1), but we do not anticipate that any fundamental difficulties lie in the way.

7.4.2 Higher-order polymorphism

Our subtyping algorithm has been smoothly extended to support the higher-order polymorphic type system of Haskell [Jon93]. Two design issues have arisen directly out of the implementation work: (1) how should an possible subtyping relation at higher kinds be defined, and (2) what assumptions should be made regarding the variance of higher-order type variables?

The first question could be answered by prescribing just the trivial subtype relation; this would be sufficient for making all Haskell programs accepted by the algorithm. However, it turns out that the flexible typings of **do**-expressions we have assumed in section 2.5.5 can be quite conveniently obtained if a higher-order relation between monads can also be expressed; i.e. if the informal fact `Cmd < O s` can also be used by the compiler. For this reason we have devised a scheme where subtype relations at higher kinds are *inferred* at need, on basis of the ground subtype axioms given by the programmer. The inference rule which governs this scheme can be explained as follows:

$$\frac{\tau\ \alpha <_P \rho\ \alpha \quad \alpha \notin fv(\tau, \rho)}{\tau <_P \rho}$$

That is, $\tau <_P \rho$ iff $\tau\ \alpha <_P \rho\ \alpha$ *for all* α, which is the definition of higher-order subtyping one also finds in most theoretical studies of the subject, e.g. [PS97].

The second question is quite naturally answered by the most pessimistic assumption; that is, arbitrary higher-order type variables are assumed to be invariant in their arguments. In [Ste97] a higher-order system with *sub-kinding* is described, which allows type variables to be parametric in their variance. However, that seems to be an overly advanced feature for a system like ours that does not even infer subtype constraints. Still, our experience with the higher-order subtyping feature of our language is currently quite limited, so we cannot really say for sure that our choice is the right one.

In any case, the separate handling of depth subtyping in our system still makes it possible to solve subtyping problems like

$$\{\alpha\ \tau \leq\ t\ \rho\}$$

by taking variances into account. Assuming t is covariant, our algorithm breaks down this problem into the sub-problem

$$\{\alpha \leq\ t, \tau \leq\ \rho\}$$

on the grounds that if Φ is a solution to this problem, we can actually derive both

$$\emptyset \vdash_P \Phi(\alpha\ \tau) \leq \Phi(t\ \tau)$$

and

$$\emptyset \vdash_P \Phi(t\ \tau) \leq \Phi(t\ \rho)$$

without making any assumptions on α whatsoever.

7.4.3 Pattern-matching

The definition of functions using general pattern-matching is a rather complex feature in Haskell as in most other functional languages, and its compilation into something similar to the datatype selection expressions we provide in our calculus generally requires a quite substantial effort by a language implementation [PJ87, Aug87]. Moreover, for the sake of comprehensible diagnostics in the case of type errors, Hugs and other common functional programming systems perform type-checking/type-inference on their high-level input, rather than on the transformed programs.

In the context of ordinary Hindley/Milner type inference, deriving inference rules for pattern-matching function definitions is quite easy, given rules that handle simple case- and lambda-expressions. But since we are dealing with type inequalities, which we furthermore attempt to solve at the earliest stage possible, the derivation of high-level rules equivalent to our basic algorithm turns out to be a rather tricky task.

Faced with this problem we basically have two choices. Either we put a considerable effort into the development a high-level algorithm that has exactly the same behaviour as the one in section 5.5 would have on transformed programs (for the benefit of those programmers who understand pattern-matching in terms of the formal translation). Or we try to find an approximation that is both easier to describe in high-level language terms, and easier to implement. We have gone for the second alternative, and our approximation can informally be described as follows:

1. First a type for each pattern is inferred, and a least upper bound for each pattern-column is computed, yielding a skeleton type for the function.

2. Then the type of each right-hand side is inferred, with the corresponding pattern variables in scope.

3. Then the constraints reachable from the type of any non-trivial pattern-column are solved (by non-trivial we mean a pattern-column that contains at least one constructor).

4. Then, finally, a least upper bound for the function types of all branches is computed.

We have found that this approximation gives the inference algorithm a quite intuitive behaviour (i.e. it seems to fail only when there really is an error, and

the error messages are moreover quite reasonable), and it has also the merit of only deviating from the algorithm in section 5.5 when there are overlapping patterns.[2] We have not yet encountered any example where this implementation has any negative consequences.

7.4.4 Refined inference for derived constructs

We saw in section 5.5 that the contextual flow during type-checking of an application was from the argument types towards the type of the function. We have chosen this strategy on a mostly pragmatic basis, since without further analysis of the expressions at hand, it is not clear which particular flow between arguments and function that would actually give the best result.

However, the high-level O'Haskell syntax contains several examples of constructs that semantically are just applications, but where the ideal flow of contextual type-information is manifest. These examples include **case**-expressions, generator statements (i.e. commands of the form $p \leftarrow e$), and **forall**-loops. In all these cases, we find lambda-bound variables for which there is additional information available, namely an expression whose type might be just the piece needed to make type inference succeed for this lambda-bound variable. For this reason the O'Hugs type-checker handles some high-level constructs according to rules which are not directly derivable from the rules in figures 5.8 and 5.9, like the following one for the **forall**-loop:[3]

$$\frac{\begin{array}{c} C, D, \Gamma \models_P e : \Phi([\nu]) \\ C', D', \Gamma \cup \{x : \Phi\nu\} \models_P \textbf{do}\ cmds : \Phi'(\tau) \\ \Phi'' \models_P C' \backslash C'_\Gamma \end{array}}{\Phi''(C \cup C'_\Gamma), \Phi''(D \cup D')\Gamma \models_P \textbf{forall}\ x \leftarrow e\ \textbf{do}\ cmds : \Phi''\Phi'(\tau)}\ \textsc{ChkFor}$$

We are currently investigating whether a simple strategy can be formulated that would allow a similar kind of ideal type-information flow to be deduced for an arbitrary application expression as well.

[2] Strictly speaking, the algorithm defined in figure 5.9 cannot handle overlapping patterns since rule ChkAlts does not support a default branch. However, adding support for such branch in the formal definition would be simple (but a bit space-consuming).

[3] C.f. the translation of this command given in appendix A.

Chapter 8

Conclusions and future work

8.1 Conclusions

Most programmers are likely to have encountered guidelines and golden rules like the following every once in a while:

> *do not write side-effecting functions! . . . avoid type-casts! . . . protect critical sections! . . . encapsulate state-variables! . . . assume no order on user input! . . .*

The exact wordings may of course differ, but it is probably no overstatement to say that the essence of these rules forms a fair part of what is generally considered to be a "good programming style" in most communities.

The mere existence of programming virtues in the areas above reveals some important information: programming languages in general have failed to help the programmer to uphold the corresponding properties. As a contrasting example, one hardly sees admonitions like

> *do not write self-modifying code!*

anymore, simply because virtually all programming languages prohibit such practices in favour of more hygienic programming techniques. One conclusion one can draw from this discrepancy is that programming language designers obviously do not believe in the full validity of the programming guidelines quoted here above!

In this dissertation we have argued that declarative properties, type safety, concurrent objects, and reactivity *can* be integrated and actively enforced in a practically useful language, and that implementing this language does not require any fundamentally new techniques. However, we have also been careful to point out that simply prohibiting the symptoms of a "bad programming style" is not a sufficient solution — the history of programming languages teaches us

that when a dubious language feature is removed, the real nature of the so called "urgent situations" that allegedly has motivated it tend to pop up to the surface. So has, for example, the removal of the *goto* statement created new insights in the need for exception mechanisms in modern languages. Similarly, the removal of general assignments in functional languages has given us monads. Our goal has been to design a language with these lessons in mind.

Specifically, we have presented the programming language *O'Haskell*, whose characteristics can be restated as follows:

- Side-effecting *expressions*, but notably not side-effecting *commands*, are prohibited.
- Type casts are disallowed, but typing flexibility is enhanced by the combination of both subtyping and parametric polymorphism.
- Protection of critical sections is enforced, albeit implicitly through other language constructs.
- State-variables are restricted to be encapsulated within objects, but first-class status is instead given to their access methods.
- Selective method filtering or active blocking is not allowed, yet synchronous method calls are provided for bi-directional data exchange between concurrent objects.

We have also described a full-scale implementation of the language, and illustrated its usefulness on a wide range of programming problems. We summarize our results, and draw our conclusions below.

- Monadic programming has been successfully extended to the realm of concurrent object-oriented languages. We have shown how assignments, state-encapsulation, concurrency, and message-passing communication can be added to a purely functional language without breaking its declarative characteristics. This result simultaneously illustrates that functional concepts like referential transparency, equational reasoning and higher-order functions can be smoothly integrated in a concurrent object-oriented language.
- Reactivity has been shown to be a natural property of concurrent objects. The core idea behind our approach is to ban active, blocking input commands, in order to obtain programs that *react* to, rather than request, external events. A simple semantics for asynchronous, message-passing reactive objects has been defined, which guarantees that in the absence of infinite looping and deadlock, any object will eventually reach a state where it can react to any new message submitted to it. We have shown that this semantics is quite compatible with established object-oriented programming practice, and that problems which seem to call for indefinitely blocking method calls can be conveniently solved by abstraction over callback methods.

- A simple yet powerful type system has been introduced, that supports both parametric polymorphism and subtyping based on a primitive notion of name-inequivalence. Haskell-style algebraic datatypes, as well as records in the style of Java interfaces, can be defined and extended by the programmer, and the subtyping relation supports both polymorphic type extension and variance-based depth subtyping for parameterized types. The type system has been proven sound relative to the dynamic semantics of the language, and its theoretical properties, notably the existence of principal types, have been investigated.

- The type system has been supplemented by a partial type inference algorithm which trades in generality for readability in the sense that it always infers constraint-free types if it succeeds. The question of a declarative specification of the limits of the algorithm is left open, but we have shown that the algorithm is complete w.r.t. the basic type systems of both Haskell and Java, and that experimental results suggest that it finds types for approximately all constraint-free typing problems that occur in practice. We have also described how type-checking in the context of this algorithm can be implemented, and we conclude that partial type inference is indeed a viable alternative to approaches that either ban the combination of subtyping and polymorphism, or require the use of type annotations throughout.

- The full-scale implementation of the language verifies that it can be realized using established implementation techniques, and that compilation is as fast and straightforward as can be expected for any functional language. In particular, the polymorphic subtype inference algorithm exhibits the same linear-in-practice behaviour as the standard inference algorithm utilized in most functional languages.

The supplied coding exercises furthermore support our initial claim that the programming style of O'Haskell is well-adapted for the task of modern software construction. Concurrent, communicating, and reactive software components can be expressed with ease, as can purely algorithmic tasks. Our concluding remark must be that a combination of the functional, object-oriented, and concurrent paradigms need not lead to a contorted language with overly many and conflicting features. We consider O'Haskell to be a simple, well-balanced language where the pieces fit together as a whole, and we believe that this shows in the programming examples we have given.

As we have stated in the introduction to this thesis, striving for *simplicity* has been a major goal for our work. Another recurring theme might have been even easier to spot: an acknowledgement of the *dualities* that lie inherent in many aspects of programming language design. Hence we can find support for values *and* objects in O'Haskell, as well as for functions *and* processes, datatypes *and* records, polymorphism *and* subtyping, etc. In all these areas, numerous examples of purification in one direction or the other are easy to find. We have wanted to show that a combination of seemingly opposing concepts does

not need to imply that the result must be inconsistent, or the concepts must get compromised beyond recognition — it is often the case that opposites fit together quite nicely by their true nature. This can preferably be read as a general statement of belief from the author, with applicability to more areas in life than just computer science.

8.2 Future work

Future directions for this work can be envisaged on three different levels.

Practical developments This level includes continued work on the interpreter, the development of a compiled implementation, and realization of the improved implementation techniques we have outlined in chapter 7. We also have some current plans for a version of the language that is based on a *call-by-value* calling convention, instead of the current call-by-need semantics inherited from Haskell. Such an implementation would bring O'Haskell a bit closer to the mainstream of programming languages, which would facilitate a more direct comparison of performance issues between O'Haskell and other concurrent and/or object-oriented language implementations.

First and foremost, though, we would like to see the language tested and evaluated on a larger scale. We are specifically interested in defining user-interface libraries and application frameworks, since this is one area where we expect the strengths of O'Haskell to enable much clearer and more intuitive interface structures than what can be found in contemporary designs. Our current work on an interface to the graphical toolkit Tk seems to verify this assumption. We would also like to gather some statistical information on how the type inference algorithm behaves when used in a larger setting, that perhaps involves programmers who have no prior experience with languages that support type inference, and thus have less reason to expect a particular behaviour.

Theoretic developments Here our primary interest lies in finding better characterizations of which expressions our type inference algorithm accepts. Even if a declarative specification that is both exact and simple may be hard to find, we believe that much better approximations must be obtainable. We have already mentioned the basic type system of Pizza as a possible lower bound, another possibility would be to investigate the relationship to Pierce and Turner's declarative algorithm specification in more detail. The algorithm itself is of course not frozen either; we have indicated a possible extension in section 7.4.4, and practical experience will no doubt provide motivation for further fine-tuning.

In any case, the full implementation of our type-checker, including the higher-order generalization of our subtyping system, will need to be formally defined. This is especially important in our case since in the absence of an exact specification, the algorithm itself is actually part of the language definition.

For the dynamic aspects of our language, a semantic theory of equivalence for programs would certainly be valuable, especially for the purpose of gaining

better insight in how the reactive semantics of O'Haskell relates to other concurrency formalisms. Work by Jeffrey, among others, could probably be used as a starting point in this respect [Jef95].

Development of the language Although we consider the basic language design to be pretty stable by now, there are nevertheless a number of interesting extensions that we would like to explore. One such issue is simple and has already been touched, and it amounts to extending the type system with implicit local quantification according to the ideas described in [Läu92, Jon96b, Jon97]. We have also indicated that the syntax for record construction/template formation might need to be reconsidered, in order to simplify the practical aspects of software reuse through parameterization and redirection (see section 4.7).

Another extension concerns the support for recursive types. Such a move should not have to affect the basic type system in any significant way, but the constraint solver would probably need some major restructuring in order to be able to cope with recursive constraints. We are not sure, though, whether this can be accomplished without breaking other more important properties of the constraint solver.

However, the most rewarding extension would actually imply a reduction of concepts. The established definition of the overloading system of Haskell is in terms of a type-directed source-to-source translation, which replaces overloaded expressions with functions taking additional *dictionary* arguments containing concrete implementations of the primitive overloaded operators [WB89]. In O'Haskell, these dictionary values could preferably be represented as records, and the overloaded operators as record selectors. A type class would then become nothing more than a record type declaration (see also [Jon97] for some developed thoughts in this direction).

The extension we are considering seeks to actually eliminate the distinction between type classes and record types in the full source language, so that overloading and parameterization w.r.t. a record become equivalent operations. The goal here is to be able to use the power of overloading and implicit dictionary handling wherever this is appropriate, while having the option of inserting an explicit dictionary record if the programming task so demands. Not only would such a scheme remove the confusing overlap between type classes and records that O'Haskell currently suffers from; it would also mean that many of the seemingly arbitrary restrictions that surround type classes in Haskell could be lifted [JJM97]. We do however anticipate that some special syntax for dictionary application will be needed to carry this through.

Appendix A

Translation of extended do-syntax

do **if** e **then** {$cmds_1$}; **else** {$cmds_2$}; $cmds$
= **do** (**if** e **then** **do** $cmds_1$ **else** **do** $cmds_2$); $cmds$

do **if** e **then** {$cmds_1$}; $cmds$
= **do** (**if** e **then** **do** $cmds_1$ **else** done); $cmds$

do **case** e **of** {p_1 -> $cmds_1$; ... ; p_n -> $cmds_n$}; $cmds$
= **do** **case** e **of** {p_1 -> **do** $cmds_1$; ... ; p_n -> **do** $cmds_n$}; $cmds$

do **forall** p <- e **do** {$cmds_1$}; $cmds$
= **do** forall_ (\p -> **do** $cmds_1$) e; $cmds$

do **while** e **do** {$cmds_1$}; $cmds$
= **do** while_ (**do** return e) (**do** $cmds_1$); $cmds$

do **fix** {p_1 <- e_1 ; ... ; p_n <- e_n}; $cmds$
= **do** p <- fix_ (\p -> **do** p_1 <- e_1 ; ... ; p_n <- e_n; return p); $cmds$
where $p = (x_1, \ldots, x_m)$, and $x_1 \ldots x_m = fv(p_1) \uplus \ldots \uplus fv(p_n)$

do $cmds$ **handle** {p_1 -> $cmds_1$; ... ; p_n -> $cmds_n$}
= handle_ (**do** $cmds$) (\x -> **case** x **of**
{p_1 -> **do** $cmds_1$; ... ; p_n -> **do** $cmds_n$; _ -> raise x})
where x is a new variable

```
class Monad m => FixMonad m where
   fix_    :: (a -> m a) -> m a


class Monad m => ExceptionMonad m where
   raise   :: Error -> m a
   handle_ :: m a -> (Error -> m a) -> m a


forall_         :: Monad m => (a -> m b) -> [a] -> m ()
forall_ f []     = done
forall_ f (x:xs) = do f x
                      forall_ f xs


while_          :: Monad m => m Bool -> m a -> m ()
while_ pred cmd  = do cont <- pred
                      if cont then
                         cmd
                         while_ pred cmd
```

Appendix B

Alternative encoding of the telecom example

```
struct S =
   off_hook' :: O S ()
   on_hook'  :: O S ()
   digit'    :: Digit -> O S ()
   answered' :: O S ()
   timeout'  :: Time -> O S ()
   cleared'  :: O S ()
   seize'    :: PotsA -> O S Bool

pots' myaddr tele_os =
   template
      state := let

         off_hook'   = done
         on_hook'    = done
         digit' _    = done
         answered'   = done
         timeout' _  = done
         cleared'    = done
         seize' _    = return False

         idle = struct

            off_hook' = do
               tele_os.start_tone myaddr Dial
               state := getNumber []
```

```
   seize' p = do
      tele_os.start_ring myaddr
      state := incoming p
      return True

   ..S

getNumber nr = struct

   on_hook' = do
      if null nr then
         tele_os.stop_tone myaddr
      state := idle

   digit' d = do
      if null nr then
         tele_os.stop_tone myaddr
      let nr' = nr ++ [d]
      resp <- tele_os.analyse nr'
      case resp of
         Invalid ->
            tele_os.start_tone myaddr Fault
            state := waitOnHook Fault
         MoreDigits ->
            state := getNumber nr'
         Valid p_b adr ->
            let me = struct
                        cleared = action
                           state.cleared'
                        answered = action
                           state.answered'
            ready <- do p_b.seize me
                     handle Deadlock -> return False
            if ready then
               tele_os.start_tone myaddr Ring
               tag <- tele_os.start_timer (Sec 90)
                      (\tag ->
                       action state.timeout' tag)
               state := ringing p_b adr tag
            else
               tele_os.start_tone myaddr Busy
               state := waitOnHook Busy

   ..S
```

```
ringing p adr tag = struct

   on_hook' = do
      p.cleared
      tele_os.stop_tone myaddr
      state := idle

   answered' = do
      tele_os.stop_tone myaddr
      tele_os.connect myaddr adr
      state := speech p (Just adr)

   timeout' t
      | t == tag = do
         p.cleared
         tele_os.stop_tone myaddr
         tele_os.start_tone myaddr Fault
         state := waitOnHook Fault
      | otherwise =
         done

   ..S

incoming p = struct

   off_hook' = do
      tele_os.stop_ring myaddr
      p.answered
      state := speech p Nothing

   on_hook' = do
      state := idle

   cleared' = do
      tele_os.stop_ring myaddr
      state := idle

   ..S

speech p Nothing = struct

   on_hook' = do
      p.cleared
      state := idle
```

```
            cleared' = do
               state := waitOnHook None

            ..S

         speech p (Just adr) = struct

            on_hook' = do
               p.cleared
               tele_os.disconnect myaddr adr
               state := idle

            cleared' = do
               tele_os.disconnect myaddr adr
               state := waitOnHook None

            ..S

         waitOnHook tone_type = struct

            on_hook' = do
               if tone_type /= None then
                  tele_os.stop_tone myaddr
               state := idle

            ..S

      in idle

   in (
      struct
         off_hook = action state.off_hook'
         on_hook  = action state.on_hook'
         digit d  = action state.digit' d ,
      struct
         cleared  = action state.cleared'
         seize p  = request state.seize' p )
```

Appendix C

Self-stabilizing IO automata

Each p belongs to the packet alphabet P.
The state of the automaton consists of a single variable $Q_{u,v} \in P \cup \{nil\}$.

$\textsc{Send}_{u,v}(p)$ $(*$ input action $*)$
  Effect: If $Q_{u,v} = nil$ then $Q_{u,v} := p$

$\textsc{Free}_{u,v}$ $(*$ output action $*)$
  Precondition: $Q_{u,v} = nil$
  Effect: None

$\textsc{Receive}_{u,v}(p)$ $(*$ output action $*)$
  Precondition: $p = Q_{u,v} \neq nil$
  Effect: $Q_{u,v} := nil$

Figure C.1: Unit Storage Data Link automaton

State:
$queue_u[v] \in P$
$free_u[v] \in \{true, false\}$

$\text{SEND}_{u,v}(p)$ $(*$ output action $*)$
Precondition: $free_u[v] = true$ and p is head of $queue_u[v]$
Effect: $free_u[v] := false$; remove p from head of $queue_u[v]$

$\text{FREE}_{u,v}$ $(*$ input action $*)$
Effect: $free_u[v] := true$

$\text{RECEIVE}_{u,v}(p)$ $(*$ input action $*)$
Effect: any, except on $queue_u[v]$ and $free_u[v]$

Figure C.2: Original stabilizable node automaton

Let $P = P_{data} \cup P_{req} \cup P_{resp}$

State:
  $queue_u[v] \in P$
  $free_u[v] \in \{true, false\}$
  $count_u[v] \in \{0..3\}$
  $mode_u[v] \in \{reset, snapshot\}$
  $phase_u[v] \in \{true, false\}$
  $freeq_u[v] \in \{true, false\}$
  $turn_u[v] \in \{response, data\}$

$\textsc{Send}_{u,v}(p)$ ($*$ output action for $p \in P_{data}$ only $*$)
  Preconditions:
    $freeq_u[v] = true$ and $free_u[v] = true$
    p is head of $queue_u[v]$
    ($l(u,v) = u$ and $phase_u[v] = false$) or ($l(u,v) = v$ and $turn_u[v] = data$)
  Effect:
    $freeq_u[v] := false$ and $free_u[v] := false$
    Remove p from head of $queue_u[v]$
    $turn_u[v] := response$
    $phase_u[v] := true$

$\textsc{Free}_{u,v}$ ($*$ input action $*$)
  Effect: $freeq_u[v] := true$ and $free_u[v] := true$

$\textsc{Receive}_{u,v}(p)$ ($*$ input action for $p \in P_{data}$ only $*$)
  Effect: same as before

$\textsc{Send}_{u,v}(p_{req})$ ($*$ output action $*$)
  Preconditions:
    $l(u,v) = u$
    $phase_u[v] = true$ or $queue_u[v]$ is empty
    $freeq_u[v] = true$
    $p_{req}.count = count_u[v]$
    $p_{req}.mode = mode_u[v]$
  Effect:
    $freeq_u[v] := false$
    $phase_u[v] := true$

Figure C.3: Node automaton after stabilizing transformation

$\textsc{Receive}_{u,v}(p_{req})$ $(*$ input action $*)$
Effect:
If $p_{req} \neq count_u[v]$ and $l(u,v) = v$ then
$count_u[v] := p_{req}.count$
$mode_u[v] := p_{req}.mode$

$\textsc{Send}_{u,v}(p_{resp})$ $(*$ output action $*)$
Preconditions:
$l(u,v) = v$
$turn = response$ or $queue_u[v]$ is empty
$freeq_u[v] = true$
$p_{resp}.count = count_u[v]$
If $mode_u[v] = snapshot$ then $p_{resp}.state = s|u$ else $p_{resp}.state = f(s|u,v)$
Effect:
If $mode_u[v] = reset$ then $s|u := f(s|u,v)$
$mode_u[v] := snapshot$
$turn_u[v] := data$
$freeq_u[v] := false$

$\textsc{Receive}_{u,v}(p_{resp})$ $(*$ input action $*)$
Effect:
If $count_u[v] = p_{resp}.count$ and $phase_u[v] = true$ and $l(u,v) = u$ then
If $mode_u[v] = snapshot$ then
If $(s|u, nil, nil, p_{resp}.state) \notin L_{u,v}$ then $mode_u[v] = reset$
Else If $mode_u[v] = reset$ then $s|u = f(s|u,v)$
$phase_u[v] := false$
$count_u[v] := (count_u[v] + 1) \bmod 4$

Figure C.4: Node automaton after stabilizing transformation (continued)

Appendix D

A stabilizable global reset protocol

```
struct Resettable =
   reset :: Action

struct ResetNetwork a < Network a, Resettable

struct ResetClient a < Client a, Resettable

data ResetMsg a = DataMsg  a
                | AbortMsg Int
                | AckMsg
                | ReadyMsg

data ResetMode = AbortMode
               | ConvergeMode
               | ReadyMode

struct LocalState =
    mode                     :: ResetMode
    you_parent, you_pending  :: Bool
    ack_in_q, rdy_in_q       :: Bool
    distance                 :: Int

global_reset ::
   Int ->
   (ResetClient a, NullNetwork) ->
   Template (ResetNetwork a,
             StabilizableClient (ResetMsg a) LocalState)
```

```
global_reset dim = \(client,network) ->
   template
      ack_pend := []
      parent   := Nothing
      distance := 0
      buffer   := array' (1,dim) []
      queue    := array' (1,dim) []
   in let
      mode [] Nothing  = ReadyMode
      mode [] (Just _) = ConvergeMode
      mode _  _        = AbortMode

      send i m = action
         if mode ack_pend parent == ReadyMode then
            enqueue i (DataMsg m)

      enqueue i m = do
         queue!i := queue!i ++ [m]

      poll i = request
         case queue!i of
            m:ms ->
               queue!i := ms
               return (Just m)
            [] ->
               return Nothing

      receive i m = action
         do_receive i m

      do_receive i (DataMsg m) = do
         if mode ack_pend parent == ReadyMode then
            client.receive i m
         else
            buffer!i := buffer!i ++ [m]

      do_receive i (AbortMsg dist) = do
         if mode ack_pend parent == ReadyMode then
            propagate (Just i) dist
         if dist == 0 || mode ack_pend parent /= ReadyMode then
            enqueue i AckMsg
         buffer!i := []

      do_receive i AckMsg = do
         if i `elem` ack_pend then
            ack_pend := delete i ack_pend
```

```
      case mode ack_pend parent of
         ConvergeMode ->
            enqueue (the parent) AckMsg
         ReadyMode ->
            reset_completed
         AbortMode ->
            done

do_receive i ReadyMsg = do
   if mode ack_pend parent == ConvergeMode
    && parent == Just i then
      parent := Nothing
      reset_completed

reset_completed = do
   client.reset
   forall j <- [1..dim] do
      enqueue j ReadyMsg
   forall i <- [1..dim] do
      forall m <- buffer!i do
         client.receive i m
      buffer!i := []

propagate par dist = do
   parent := if dist /= 0 then par else Nothing
   distance := dist
   forall j <- [1..dim] do
      enqueue j (AbortMsg (distance+1))
   ack_pend := [1..dim]

reset = action
   propagate Nothing 0

current_state i = request
   return (struct
      mode        = mode ack_pend parent
      you_parent  = parent == Just i
      you_pending = i `elem` ack_pend
      ack_in_q    = AckMsg `elem` queue!i
      rdy_in_q    = ReadyMsg `elem` queue!i
      distance    = distance)

check_invariants i peer = request
   let p0  = i `elem` ack_pend
      p1  = AbortMsg (distance+1) `elem` queue!i
      p2  = peer.mode == AbortMode && peer.you_parent
```

```
            p3  = peer.ack_in_q
            p4  = parent == Just i
            p5  = mode ack_pend parent == ConvergeMode
            p6  = AckMsg `elem` queue!i
            p7  = not peer.you_pending && peer.mode /= ReadyMode
            p8  = peer.rdy_in_q
            p9  = f1 (queue!i)
            p10 = mode ack_pend parent == ReadyMode
            p11 = distance == peer.distance + 1

            f1 []          = False
            f1 [ReadyMsg]  = True
            f1 [DataMsg _] = True
            f1 (m:ms)      = f1 ms

            f2 [AbortMsg _,AckMsg]          = True
            f2 [AckMsg,ReadyMsg,AbortMsg _] = True
            f2 [ReadyMsg,AbortMsg _]        = True
            f2 (AckMsg:ReadyMsg:ms)         = f3 ms
            f2 (ReadyMsg:ms)                = f3 ms
            f2 ms                           = f3 ms
            f3 []             = True
            f3 (DataMsg _:ms) = f3 ms
            f3 _              = False

            pA = p0 == (p1 || p2 || p3)
            pB = length (filter id [p1,p2,p3]) <= 1
            pC = (p4 `implies` p5) == (p6 || p7 || p8)
            pD = length (filter id [p6,p7,p8]) <= 1
            pR = p9 `implies` p10
            pP = p4 `implies` (p11 || p8)
            pQ = f2 (queue!i)

            a `implies` b = if a then b else True

         return (pA && pB && pC && pD && pR && pP && pQ)

      local_reset i = action
         do_receive i AckMsg
         do_receive i ReadyMsg
         if parent == Just i then
            parent := Nothing
         buffer!i := []
         queue!i  := []

in (struct ..ResetNetwork, struct ..StabilizableClient)
```

Appendix E

Proofs

E.1 Proof of lemma 5.3.8

Assume $C, \Gamma \models_P e : \tau$, $\alpha \in fv(\tau)$, *and* $\beta \in fv(C)$. *Then:*

1. α *is a "new" variable, distinct from any variable in the context of the derivation.*

2. *either* $\beta \in fv(\Gamma)$, *or* β *is a "new" variable, distinct from any variable in the context of the derivation.*

3. C *is a* Γ*-constraint.*

Proof. By structural induction on e. At each occurrence, let *new* stand for a set of variables that is new in the sense given above.

Case (x)**.** The derivation must be of the form

$$\frac{\overline{\beta_i} = fv(\tau) \quad C = \{\nu_i \leq \beta_i\}^{\beta_i \in (\tau^- \setminus \overline{\alpha})} \cup \{\beta_i \leq \nu_i\}^{\beta_i \in (\tau^+ \setminus \overline{\alpha})}}{C, \Gamma \models_P x : [\overline{\nu_i}/\overline{\beta_i}]\tau} \textsc{InfVar}$$

where $\Gamma = \Gamma' \cup \{x : \forall \overline{\alpha}.\tau\}$. By definition $\overline{\nu_i} \subseteq new$, and since $(\tau^- \setminus \overline{\alpha}) \cup (\tau^+ \setminus \overline{\alpha}) = fv(\forall \overline{\alpha}.\tau)$, we have

$$\begin{array}{l} fv([\overline{\nu_i}/\overline{\beta_i}]\tau) \subseteq new \\ fv(C) \subseteq fv(\Gamma) \cup new \\ C \text{ is a } \Gamma\text{-constraint} \end{array}$$

as required.

Case $(e\ e')$**.** The derivation must be of the form

$$\frac{C, \Gamma \models_P e : \tau \quad C_i, \Gamma \models_P e_i : \tau_i \quad \Phi \models_P \{\tau \leq \overline{\tau_i} \to \nu\}}{\Phi(C \cup \bigcup_i C_i), \Gamma \models_P e\ \overline{e_i} : \Phi\nu} \textsc{InfApp}$$

We have

$fv(\tau, \overline{\tau_i}) \subseteq new$	
$fv(C, \bigcup_i C'_i) \subseteq fv(\Gamma) \cup new$	
C and each C_i are Γ-constraints	(by induction hypothesis)
$C \cup \bigcup_i C_i$ is a Γ-constraint	(by lemma 5.3.7)
$\nu \in new$	(by definition)
$vars(\Phi) \subseteq new$	(by lemma E.5.1)

and thus

$fv(\Phi\nu) \subseteq new$	
$fv(\Phi(C \cup \bigcup_i C'_i)) \subseteq fv(\Gamma) \cup new$	
$\Phi(C \cup \bigcup_i C_i)$ is a Γ-constraint	(by lemma 5.3.7)

as required.

Case ($\lambda x \to e$). The derivation must be of the form

$$\frac{C, \Gamma' \models_P e : \tau \quad \Phi \models_P C \backslash C_\Gamma}{\Phi(C_\Gamma), \Gamma \models_P \lambda \overline{x_i} \to e : \Phi(\overline{\nu_i} \to \tau)} \textsc{InfAbs}$$

where $\Gamma' = \Gamma \cup \{x_i : \nu_i\}^i$. We have

$fv(\tau) \subseteq new$	
$fv(C) \subseteq fv(\Gamma \cup \{x_i : \nu_i\}^i) \cup new$	
C is a Γ'-constraint	(by induction hypothesis)
C_Γ is a Γ-constraint	(by lemma 5.3.7)
$fv(\{x_i : \nu_i\}^i) \subseteq new$	(by definition)
$fv(C \backslash C_\Gamma) \subseteq new$	(by definition of C_Γ)
$vars(\Phi) \subseteq new$	(by lemma E.5.1)

and thus

$fv(\Phi(\overline{\nu_i} \to \tau)) \subseteq new$	
$fv(\Phi(C \cup \bigcup_i C'_i)) \subseteq fv(\Gamma) \cup new$	
$\Phi(C_\Gamma)$ is a Γ-constraint	(by lemma 5.3.7)

as required.

Case (let $x = e$ in e'). The derivation must be of the form

$$\frac{C, \Gamma \models_P e : \tau \quad C', \Gamma' \models_P e' : \tau' \quad \Phi \models_P C' \backslash C'_\Gamma}{\Phi(C \cup C'_\Gamma), \Gamma \models_P \textbf{let } x = e \textbf{ in } e' : \Phi\tau'} \textsc{InfLet}$$

where $\Gamma' = \Gamma \cup \{x : gen(C, \tau)\}$. We have

$fv(\tau, \tau') \subseteq new$	
$fv(C) \subseteq fv(\Gamma) \cup new$	
$fv(C') \subseteq fv(\Gamma \cup \{x : gen(C, \tau)\}) \cup new$	
C is a Γ-constraint	
C' is a Γ'-constraint	(by induction hypothesis)
$C \cup C'_\Gamma$ is a Γ-constraint	(by lemma 5.3.7)
$fv(gen(C, \tau)) \subseteq fv(\tau) \subseteq new$	(by definition of gen)
$fv(C' \backslash C'_\Gamma) \subseteq new$	(by definition of C'_Γ)
$vars(\Phi) \subseteq new$	(by lemma E.5.1)

and thus

$fv(\Phi\tau) \subseteq new$	
$fv(\Phi(C \cup C'_\Gamma)) \subseteq fv(\Gamma) \cup new$	
$\Phi(C \cup C'_\Gamma)$ is a Γ-constraint	(by lemma 5.3.7)

as required.

The cases for our record and datatype extension follow the pattern for variables and applications, and will not be shown. □

E.2 Proof of theorem 5.3.11

If $C, \Gamma \models_P e : \tau$ *then* $C, \Gamma \vdash_P e : \tau$.

Proof. By structural induction on e.

Case (x). The derivation must be of the form

$$\frac{\overline{\beta_i} = fv(\tau) \quad C = \{\nu_i \leq \beta_i\}^{\beta_i \in (\tau^- \backslash \overline{\alpha})} \cup \{\beta_i \leq \nu_i\}^{\beta_i \in (\tau^+ \backslash \overline{\alpha})}}{C, \Gamma \models_P x : [\overline{\nu_i}/\overline{\beta_i}]\tau} \textsc{InfVar}$$

where $\Gamma = \Gamma' \cup \{x : \forall\overline{\alpha}.\tau\}$. Let $\Phi = [\overline{\nu_i}/\overline{\beta_i}]$ restricted to domain $\overline{\alpha}$. We have

$C, \Gamma \vdash_P x : \forall\overline{\alpha}.\tau$	(by TypVar)
$C, \Gamma \vdash_P x : \Phi\tau$	(by TypInst)
$C \vdash_P \Phi\tau \leq [\overline{\nu_i}/\overline{\beta_i}]\tau$	(by repeated use of lemma E.5.4)

and thus, by rule TypSub,

$$C, \Gamma \vdash_P x : [\overline{\nu_i}/\overline{\alpha_i}]\tau$$

as required.

Case ($e\ e'$). The derivation must be of the form

$$\frac{C, \Gamma \models_P e : \tau \quad C_i, \Gamma \models_P e_i : \tau_i \quad \Phi \models_P \{\tau \leq \overline{\tau_i} \to \nu\}}{\Phi(C \cup \bigcup_i C_i), \Gamma \models_P e\ \overline{e_i} : \Phi\nu}\ \textsc{InfApp}$$

We have

$C, \Gamma \vdash_P e : \tau$	
$C_i, \Gamma \vdash_P e_i : \tau_i$, for each i	(by induction hypothesis)
$\Phi C, \Phi\Gamma \vdash_P e : \Phi\tau$	
$\Phi C_i, \Phi\Gamma \vdash_P e_i : \Phi\tau_i$	(by lemma 5.2.1)
$\Phi(C \cup \bigcup C_i), \Phi\Gamma \vdash_P e : \Phi\tau$	
$\Phi(C \cup \bigcup C_i), \Phi\Gamma \vdash_P e_i : \Phi\tau_i$	(by lemma E.5.2)
$\emptyset \vdash_P \Phi\tau \leq \Phi(\overline{\tau_i} \to \nu)$	(by lemma 5.3.1)
$\Phi(C \cup \bigcup C_i) \vdash_P \Phi\tau \leq \Phi(\overline{\tau_i} \to \nu)$	(by lemma E.5.3)
$\Phi(C \cup \bigcup C_i), \Phi\Gamma \vdash_P e : \Phi(\overline{\tau_i} \to \nu)$	(by TypSub)
$\Phi(C \cup \bigcup_i C_i), \Phi\Gamma \vdash_P e\ \overline{e_i} : \Phi\nu$	(by repeated use of TypeApp)
$fv(\tau, \overline{\tau_i}, \nu) \cap fv(\Gamma) = \emptyset$	(by lemma 5.3.8)
$dom(\Phi) \cap fv(\Gamma) = \emptyset$	(by lemma E.5.1)

and thus

$$\Phi(C \cup \bigcup_i C_i), \Gamma \vdash_P e\ \overline{e_i} : \Phi\nu$$

as required.

Case ($\lambda x \to e$). The derivation must be of the form

$$\frac{C, \Gamma \cup \{x_i : \nu_i\}^i \models_P e : \tau \quad \Phi \models_P C \backslash C_\Gamma}{\Phi(C_\Gamma), \Gamma \models_P \lambda\overline{x_i} \to e : \Phi(\overline{\nu_i} \to \tau)}\ \textsc{InfAbs}$$

We have

$C, \Gamma \cup \{x_i : \nu_i\}^i \vdash_P e : \tau$	(by induction hypothesis)
$\Phi C, \Phi(\Gamma \cup \{x_i : \nu_i\}^i) \vdash_P e : \Phi\tau$	(by lemma 5.2.1)
$\emptyset \vdash_P \Phi(C \backslash C_\Gamma)$	(by lemma 5.3.1)
$\Phi C_\Gamma \vdash_P \Phi C$	(by lemma E.5.3)
$\Phi C_\Gamma, \Phi(\Gamma \cup \{x_i : \nu_i\}^i) \vdash_P e : \Phi\tau$	(by lemma 5.2.2)
$\Phi C_\Gamma, \Phi\Gamma \vdash_P \lambda\overline{x_i} \to e : \Phi(\overline{\nu_i} \to \tau)$	(by repeated use of TypAbs)
$fv(C \backslash C_\Gamma) \cap fv(\Gamma) = \emptyset$	(by definition of C_Γ)
$dom(\Phi) \cap fv(\Gamma) = \emptyset$	(by lemma E.5.1)

and thus

$$\Phi C_\Gamma, \Gamma \vdash_P \lambda\overline{x_i} \to e : \Phi(\overline{\nu_i} \to \tau)$$

as required.

Case (let $x = e$ in e'). The derivation must be of the form

$$\frac{C, \Gamma \models_P e : \tau \quad C', \Gamma \cup \{x : \forall\overline{\alpha}.\tau\} \models_P e' : \tau' \quad \Phi \models_P C' \backslash C'_\Gamma}{\Phi(C \cup C'_\Gamma), \Gamma \models_P \textbf{let}\ x = e\ \textbf{in}\ e' : \Phi\tau'}\ \textsc{InfLet}$$

where $\overline{\alpha} = fv(\tau) \backslash fv(C)$. We have

$C, \Gamma \vdash_P e : \tau$	
$C', \Gamma \cup \{x : \forall \overline{\alpha}.\tau\} \vdash_P e' : \tau'$	(by induction hypothesis)
$\Phi C, \Phi\Gamma \vdash_P e : \Phi\tau$	
$\Phi C', \Phi\Gamma \cup \{x : \Phi(\forall \overline{\alpha}.\tau)\} \vdash_P e' : \Phi\tau'$	(by lemma 5.2.1)
$\emptyset \vdash_P \Phi(C' \backslash C'_\Gamma)$	(by lemma 5.3.1)
$\Phi C'_\Gamma \vdash_P \Phi C'$	(by lemma E.5.3)
$\Phi C'_\Gamma, \Phi\Gamma \cup \{x : \Phi(\forall \overline{\alpha}.\tau)\} \vdash_P e' : \Phi\tau'$	(by lemma 5.2.2)
$\Phi(C \cup C'_\Gamma), \Phi\Gamma \vdash_P e : \Phi\tau$	
$\Phi(C \cup C'_\Gamma), \Phi\Gamma \cup \{x : \Phi(\forall \overline{\alpha}.\tau)\} \vdash_P e' : \Phi\tau'$	(by lemma E.5.2)
$\overline{\alpha} \cap fv(C) = \emptyset$	(by definition)
$\overline{\alpha} \cap fv(\Gamma) = \emptyset$	(by lemma 5.3.8)
$\overline{\alpha} \cap fv(\Gamma \cup \{x : \forall \overline{\alpha}.\tau\}) = \emptyset$	(by definition of fv)
$\overline{\alpha} \cap fv(C') = \emptyset$	(by lemma 5.3.8)
$\overline{\alpha} \cap vars(\Phi) = \emptyset$	(by lemma E.5.1) †
$\overline{\alpha} \cap fv(\Phi(C \cup C'_\Gamma), \Phi\Gamma) = \emptyset$	(by previous results)
$\Phi(C \cup C'_\Gamma), \Phi\Gamma \vdash_P e : \forall \overline{\alpha}.\Phi\tau$	(by TypGen)
$\Phi(C \cup C'_\Gamma), \Phi\Gamma \cup \{x : \forall \overline{\alpha}.\Phi\tau)\} \vdash_P e' : \Phi\tau'$	(by †)
$\Phi(C \cup C'_\Gamma), \Phi\Gamma \vdash_P \mathbf{let}\ x = e\ \mathbf{in}\ e' : \Phi\tau'$	(by TypLet)
$fv(C' \backslash C'_\Gamma) \cap fv(\Gamma) = \emptyset$	(by definition of C'_Γ)
$dom(\Phi) \cap fv(\Gamma) = \emptyset$	(by lemma E.5.1)

and thus

$$\Phi(C \cup C'_\Gamma), \Gamma \vdash_P \mathbf{let}\ x = e\ \mathbf{in}\ e' : \Phi\tau'$$

as required.

The cases for our record and datatype extension follow the pattern for variables and applications, and will not be shown. □

E.3 Proof of theorem 5.3.13

Assume $\Phi\Gamma \vdash^{HM} e : \forall \overline{\alpha}.\tau$, where $\overline{\alpha} \notin fv(\Phi\Gamma)$. Then $C, \Gamma \models_P e : \rho$ succeeds, and there is a Ψ such that Ψ unifies C, $\Psi\Gamma = \Phi\Gamma$, and $\Psi\rho = \tau$.

Proof. By induction on the derivation of $\Phi\Gamma \vdash^{HM} e : \forall \overline{\alpha}.\tau$.

Case (TypVar). The derivation is of the form

$$\frac{}{\Phi\Gamma \vdash^{HM} x : \Phi\forall \overline{\alpha}.\tau}\ \textsc{TypVar}$$

were $\Gamma = \Gamma' \cup \{x : \forall \overline{\alpha}.\tau\}$. W.l.o.g. we may assume that $\overline{\alpha} \cap fv(\Gamma) = \emptyset$, since α-equivalent type schemes are identified. Hence we can also assume $\overline{\alpha} \cap dom(\Phi) = \emptyset$, since this can always be achieved by restriction. Now let

$$\begin{aligned} &\overline{\beta_i} = fv(\tau) \\ &C = \{\nu_i \leq \beta_i\}^{\beta_i \in (\tau^- \backslash \overline{\alpha})} \cup \{\beta_i \leq \nu_i\}^{\beta_i \in (\tau^+ \backslash \overline{\alpha})} \\ &\Psi = \Phi \circ [\overline{\beta_i}/\overline{\nu_i}] \end{aligned}$$

and we have

$$C, \Gamma \models_P x : [\overline{\nu_i}/\overline{\beta_i}]\tau \text{ succeeds}$$

by InfVar, and also

$$\begin{array}{l}\Psi \textit{ unifies } C\\ \Psi\Gamma = \Phi[\overline{\beta_i}/\overline{\nu_i}]\Gamma = \Phi\Gamma\\ \Psi[\overline{\nu_i}/\overline{\beta_i}]\tau = \Phi\tau = \Phi\backslash_{\overline{\alpha}}\tau\end{array}$$

as required.

Case (TypApp). The derivation is of the form

$$\frac{\Phi\Gamma \vdash^{HM} e : \tau' \to \tau \quad \Phi\Gamma \vdash^{HM} e' : \tau'}{\Phi\Gamma \vdash^{HM} e\ e' : \tau}\ \textsc{TypApp}$$

By the induction hypothesis we have

$$\begin{array}{l}C, \Gamma \models_P e : \rho \text{ succeeds}\\ \Psi \textit{ unifies } C\\ \Psi\Gamma = \Phi\Gamma\\ \Psi\rho = \tau' \to \tau\\ C', \Gamma \models_P e' : \rho' \text{ succeeds}\\ \Psi' \textit{ unifies } C'\\ \Psi'\Gamma = \Phi\Gamma\\ \Psi'\rho' = \tau'\end{array}$$

W.l.o.g. we can assume that $dom(\Psi) \subseteq fv(C, \Gamma, \rho)$ and $dom(\Psi') \subseteq fv(C', \Gamma, \rho')$, since this can always be achieved by restriction. Thus we have $dom(\Psi) \subseteq fv(\Gamma) \cup new_1$ and $dom(\Psi') \subseteq fv(\Gamma) \cup new_2$ by lemma 5.3.8. Furthermore, since Ψ' can always be renamed so that $rng(\Psi') \subseteq fv(\Phi\Gamma, \tau') \cup new_3$, another consequence of lemma 5.3.8 is that $rng(\Psi') \cap new_2 = \emptyset$.

Let $V = fv(C, \Gamma, \rho)$, $V' = fv(C', \Gamma, \rho' \to \nu)$, and $V'' = fv(\Gamma)$. By the above arguments, $(\Psi \circ \Psi')|_V = (\Psi \circ \Psi'|_{V''})|_V$ and $(\Psi \circ \Psi')|_{V'} = (\Psi|_{V''} \circ \Psi')|_{V'}$. But since $\Psi|_{V''} = \Phi|_{V''} = \Psi'|_{V''}$, the two restricted substitutions must be equal to $(\Psi|_{V''} \circ \Psi)|_V$ and $(\Psi' \circ \Psi'|_{V''})|_{V'}$ respectively, which by the idempotence of substitutions becomes equivalent to $\Psi|_V$ and $\Psi'|_{V'}$.

We now have $[\tau/\nu]\Psi\Psi'\rho = [\tau/\nu]\Psi\rho = [\tau/\nu](\tau' \to \tau) = \tau' \to \tau$ and $[\tau/\nu]\Psi\Psi'(\rho' \to \nu) = [\tau/\nu]\Psi'(\rho' \to \nu) = [\tau/\nu](\tau' \to \nu) = \tau' \to \nu$. That is, $[\tau/\nu] \circ \Psi \circ \Psi'$ *unifies* $\{\rho \leq \rho' \to \nu\}$. From lemma 5.3.3 it then follows that $\Phi' \models_P \{\rho \leq \rho' \to \nu\}$ succeeds, and that there is a Φ'' such that $[\tau/\nu] \circ \Psi \circ \Psi' = \Phi'' \circ \Phi'$. Thus we know that

$$\Phi'(C \cup C'), \Gamma \models_P e\ e' : \Phi\nu \text{ succeeds}$$

by rule InfApp. Moreover, by letting $\Psi'' = [\tau/\nu] \circ \Psi \circ \Psi'$ and utilizing the idempotence of substitutions we also have

$$\begin{array}{l}\Psi''\Phi'C = \Phi''\Phi'\Phi'C = [\tau/\nu]\Psi\Psi'C = [\tau/\nu](\Psi \circ \Psi')|_V C = [\tau/\nu]\Psi C = \Psi C\\ \Psi''\Phi'C' = \Phi''\Phi'\Phi'C' = [\tau/\nu]\Psi\Psi'C' = [\tau/\nu](\Psi \circ \Psi')|_{V'} C' = [\tau/\nu]\Psi'C' = \Psi'C'\\ \Psi''\Gamma = [\tau/\nu]\Psi\Psi'\Gamma = \Psi\Gamma\\ \Psi''\nu = [\tau/\nu]\Psi\Psi'\nu = [\tau/\nu]\nu = \tau\end{array}$$

and thus

$$\begin{array}{l}\Psi'' \textit{ unifies } \Phi'(C \cup C') \\ \Psi''\Gamma = \Phi\Gamma \\ \Psi''\Phi'\nu = \tau\end{array}$$

as required.

Case (TypAbs). The derivation is of the form

$$\frac{\Phi\Gamma \cup \{x : \tau'\} \vdash^{HM} e : \tau}{\Phi\Gamma \vdash^{HM} \lambda x \to e : \tau' \to \tau}\ \textsc{TypAbs}$$

Since ν is new by definition, we have that $\Phi\Gamma \cup \{x : \tau'\} = [\tau'/\nu]\Phi(\Gamma \cup \{x : \nu\})$, and so, by the induction hypothesis

$$\begin{array}{l}C, \Gamma \cup \{x : \nu\} \models_P e : \rho \text{ succeeds} \\ \Psi \textit{ unifies } C \\ \Psi\Gamma = [\tau'/\nu]\Phi\Gamma \\ \Psi\nu = [\tau'/\nu]\Phi\nu \\ \Psi\rho = \tau\end{array}$$

Moreover, since C_Γ and $C \backslash C_\Gamma$ are subsets of C, Ψ also unifies C_Γ and $C \backslash C_\Gamma$. Hence we know that $\Phi' \models_P C \backslash C_\Gamma$ succeeds, and there is a Φ'' such that $\Psi = \Phi'' \circ \Phi'$ (lemma 5.3.3). By rule InfAbs it now follows that

$$\Phi'(C_\Gamma) \models_P \lambda x \to e : \Phi(\nu \to \rho) \text{ succeeds}$$

Furthermore, by utilizing the idempotence of substitutions, we also have $\Psi\Phi'(C_\Gamma) = \Phi''\Phi'\Phi'(C_\Gamma) = \Psi(C_\Gamma)$ and thus

$$\begin{array}{l}\Psi \textit{ unifies } \Phi'(C_\Gamma) \\ \Psi\Gamma = [\tau'/\nu]\Phi\Gamma = \Phi\Gamma \\ \Psi\Phi'(\nu \to \rho) = \tau' \to \tau\end{array}$$

as required.

Case (TypLet). The derivation is of the form

$$\frac{\Phi\Gamma \vdash^{HM} e : \forall\overline{\alpha}.\tau \quad \Phi\Gamma \cup \{x : \forall\overline{\alpha}.\tau\} \vdash^{HM} e' : \tau'}{\Phi\Gamma \vdash^{HM} \textbf{let } x = e \textbf{ in } e' : \tau'}\ \textsc{TypLet}$$

By the induction hypothesis we have

$$\begin{array}{l}C, \Gamma \models_P e : \rho \text{ succeeds} \\ \Psi \textit{ unifies } C \\ \Psi\Gamma = \Phi\Gamma \\ \Psi\rho = \tau\end{array}$$

Here we can assume that $dom(\Psi) \subseteq fv(C, \Gamma, \rho)$, since this can always be achieved by restriction. Thus we have $dom(\Psi) \subseteq fv(\Gamma) \cup new_1$, and also $fv(\rho) \subseteq new_1$, by lemma 5.3.8. Another consequence of the same lemma is that $fv(\rho) \cap dom(\Phi) = \emptyset$.

Let $\overline{\beta} = fv(\rho) \backslash fv(C)$, that is, $gen(C, \rho) = \forall\overline{\beta}.\rho$. In order to proceed we need to establish that $\emptyset \vdash_P \Psi gen(C, \rho) \leq \forall\overline{\alpha}.\Psi\rho$. This can be shown using lemma E.5.6 together with the following observations:

1. $\Psi\forall\overline{\beta}.\rho = \forall\overline{\beta}.\Psi\backslash_{\overline{\beta}}\rho$
2. $\forall\overline{\alpha}.\Psi\rho = \forall\overline{\alpha}.\Psi|_{\overline{\beta}}(\Psi\backslash_{\overline{\beta}}\rho)$
3. $fv(\Psi gen(C,\rho)) \subseteq fv(\Psi C)$, and by lemma E.5.5 we have $fv(\Psi C) \subseteq fv(\Psi\Gamma) = fv(\Phi\Gamma)$. Since $\overline{\alpha} \cap fv(\Phi\Gamma) = \emptyset$ by assumption, it follows that $\overline{\alpha} \cap fv(\Psi gen(C,\rho)) = \emptyset$

We now have $\Phi\Gamma \cup \{x : \forall\overline{\alpha}.\Psi\rho\} \vdash_P e' : \tau'$ and $\emptyset \vdash_P \Psi gen(C,\rho) \leq \forall\overline{\alpha}.\Psi\rho$, and so by proposition 5.2.4 we also have a derivation of $\Phi\Gamma \cup \{x : \Psi gen(C,\rho)\} \vdash_P e' : \tau'$. But $\Phi\Gamma \cup \{x : \Psi gen(C,\rho)\}$ can also be written $\Psi\Phi(\Gamma \cup \{x : gen(C,\rho)\})$, since $\Phi\rho = \rho$ and $\Psi\Gamma = \Phi\Gamma$. Hence the induction hypothesis applies anew, and we have

$$
\begin{array}{l}
C', \Gamma \cup \{x : gen(C,\rho)\} \models_P e' : \rho' \text{ succeeds} \\
\Psi' \textit{ unifies } C' \\
\Psi'\Gamma = \Psi\Phi\Gamma \\
\Psi' gen(C,\rho) = \Psi\Phi gen(C,\rho) \\
\Psi'\rho' = \tau'
\end{array}
$$

Here we can assume that $dom(\Psi') \subseteq fv(C', \Gamma, gen(C,\rho), \rho')$, since this can always be achieved by restriction. Thus we have $dom(\Psi') \subseteq fv(\Gamma) \cup new_2 \cup fv(gen(C,\rho))$ by lemma 5.3.8. Furthermore, since Ψ' can always be renamed so that $rng(\Psi') \subseteq fv(\Psi\Phi(\Gamma \cup gen(C,\rho), \tau') \cup new_3$, another consequence of lemma 5.3.8 is that $rng(\Psi') \cap new_1 = \emptyset$.

Let $V = fv(C, \Gamma, \rho)$, $V' = fv(C', \Gamma, gen(C,\rho), \rho')$, and $V'' = fv(\Gamma, gen(C,\rho))$. By the above arguments, $(\Psi \circ \Psi')|_V = (\Psi \circ \Psi'|_{V''})|_V$ and $(\Psi \circ \Psi')|_{V'} = (\Psi|_{V''} \circ \Psi')|_{V'}$. But since $\Psi'|_{V''} = (\Psi \circ \Phi)|_{V''}$, the first substitution must be equal to $((\Psi \circ \Phi)|_{V''} \circ \Psi)|_V$. Moreover, since $dom(\Phi) \cap V'' \subseteq fv(\Gamma)$, and $\Psi\Gamma = \Phi\Gamma$, it follows that $(\Psi \circ \Phi)|_{V''} = \Psi_{V''}$. Hence, utilizing the idempotence of substitutions, we have $(\Psi \circ \Psi')|_V = (\Psi|_{V''} \circ \Psi)|_V = \Psi|_V$, and $(\Psi \circ \Psi')|_{V'} = (\Psi|_{V''} \circ \Psi')|_{V'} = ((\Psi \circ \Phi)|_{V''} \circ \Psi')|_{V'} = (\Psi'|_{V''} \circ \Psi')|_{V'} = \Psi'|_{V'}$.

Now, C'_Γ and $C'\backslash C'_\Gamma$ are both subsets of C', therefore Ψ' also unifies C'_Γ and $C'\backslash C_\Gamma$. This in turn means that $\Phi' \models_P C'\backslash C'_\Gamma$ succeeds, and there is a Φ'' such that $\Psi' = \Phi'' \circ \Phi'$ (lemma 5.3.3). By rule InfLet it follows that

$$\Phi'(C \cup C'_\Gamma), \Gamma \models_P \textbf{let } x = e \textbf{ in } e' : \Phi'\rho' \text{ succeeds}$$

Moreover, by letting $\Psi'' = \Psi \circ \Psi'$ and utilizing the idempotence of substitutions we also have

$$
\begin{array}{l}
\Psi''\Phi'C = \Psi\Psi'C = (\Psi \circ \Psi')|_V C = \Psi C \\
\Psi''\Phi'C'_\Gamma = \Psi\Psi'C'_\Gamma = (\Psi \circ \Psi')|_{V'} C'_\Gamma = \Psi' C'_\Gamma \\
\Psi''\Gamma = (\Psi \circ \Psi')|_V \Gamma = \Psi\Gamma \\
\Psi''\rho' = (\Psi \circ \Psi')|_{V'} \rho' = \Psi'\rho'
\end{array}
$$

and thus

$$
\begin{array}{l}
\Psi'' \textit{ unifies } \Phi'(C \cup C'_\Gamma) \\
\Psi''\Gamma = \Phi\Gamma \\
\Psi''\rho' = \tau'
\end{array}
$$

as required.

Case (TypGen). The derivation is of the form

$$\frac{\Phi\Gamma \vdash^{HM} e : \tau \quad \overline{\alpha} \notin fv(\Phi\Gamma)}{\Phi\Gamma \vdash^{HM} e : \forall\overline{\alpha}.\tau}\ \textsc{TypGen}$$

By the induction axiom we have

$$\begin{array}{l} C, \Gamma \models_P e : \rho \text{ succeeds} \\ \Psi \textit{ unifies } C \\ \Psi\Gamma = \Phi\Gamma \\ \Psi\rho = \tau \end{array}$$

as required.

Case (TypInst). The derivation is of the form

$$\frac{\Phi\Gamma \vdash^{HM} e : \forall\overline{\alpha}.\tau}{\Phi\Gamma \vdash^{HM} e : [\overline{\tau}/\overline{\alpha}]\tau}\ \textsc{TypInst}$$

By the induction axiom we have

$$\begin{array}{l} C, \Gamma \models_P e : \rho \text{ succeeds} \\ \Psi \textit{ unifies } C \\ \Psi\Gamma = \Phi\Gamma \\ \Psi\rho = \tau \end{array}$$

Now let $\Psi' = [\overline{\tau}/\overline{\alpha}] \circ \Psi$. Since by assumption $\overline{\alpha} \cap fv(\Phi\Gamma) = \emptyset$, we have $[\overline{\tau}/\overline{\alpha}]\Phi\Gamma = \Phi\Gamma$ and hence

$$\begin{array}{l} \Psi' \textit{ unifies } C \\ \Psi'\Gamma = \Phi\Gamma \\ \Psi'\rho = [\overline{\tau}/\overline{\alpha}]\Psi\rho = [\overline{\tau}/\overline{\alpha}]\tau \end{array}$$

as required.

Case (TypSub). Does not apply, by assumption.

The cases for our record and datatype extension follow the pattern for variables and applications, and will not be shown. □

E.4 Proof of theorem 5.3.15

If $\Gamma \vdash^J_P e : \tau$ *and* $\emptyset \vdash_P \Gamma' \leq \Gamma$*, then* $\emptyset, \Gamma' \models_P e : \rho$ *succeeds,* $fv(\rho) = \emptyset$*, and* $\emptyset \vdash_P \rho \leq \tau$*.*

Proof. By induction on the derivation of $\Gamma \vdash^J_P e : \tau$.

Case (TypVar). The derivation must now be of the form

$$\frac{}{\Gamma \vdash^J_P x : \tau}\ \textsc{TypVar}$$

where $\tau = \Gamma(x)$. Moreover, Γ' must be such that $\Gamma'(x) = \rho$, where $\emptyset \vdash_P \rho \leq \tau$. But since $fv(P) = \emptyset$, there can be no polymorphic subtype axiom in use in the derivation of $\emptyset \vdash_P \rho \leq \tau$, and so $fv(\rho) = \emptyset$. Thus we know that $\emptyset, \Gamma' \models_P x : \rho$ succeeds with the required ρ.

Case (TypApp). The derivation is of the form

$$\frac{\Phi\Gamma \vdash^J_P e : \tau' \to \tau \quad \Phi\Gamma \vdash^J_P e' : \tau'}{\Phi\Gamma \vdash^J_P e\ e' : \tau} \text{ TypApp}$$

By the induction hypothesis we have

$$\begin{array}{l} \emptyset, \Gamma' \models_P e : \rho \text{ succeeds} \\ \emptyset \vdash_P \rho \leq \tau' \to \tau \\ \emptyset, \Gamma' \models_P e' : \rho' \text{ succeeds} \\ \emptyset \vdash_P \rho' \leq \tau' \\ fv(\rho, \rho') = \emptyset \end{array}$$

It follows from lemma E.5.7 that $\emptyset \vdash_P \rho \leq \tau' \to \tau$ must imply $\emptyset \vdash_P \rho_1 \leq \tau$ and $\emptyset \vdash_P \tau' \leq \rho_2$ for some ρ_1 and ρ_2. By transitivity we also have $\emptyset \vdash_P \rho' \leq \rho_2$.

The constraint-solving problem $\rho \leq \rho' \to \nu$ thus immediately breaks down into $\rho' \leq \rho_2$ and $\rho_1 \leq \nu$. The latter problem is unifiable, hence by lemma 5.3.3 we have $[\rho_1/\nu] \models_P \{\rho_1 \leq \nu\}$. Moreover, $fv(\rho', \rho_2) = \emptyset$, so by lemma E.5.8 we also have $[\rho_1/\nu] \models_P \{\rho_2 \to \rho_1 \leq \rho' \to \nu\}$. Thus $\emptyset, \Gamma' \models_P e\ e' : \rho_1$ succeeds, with ρ_1 meeting both its requirements.

Case (TypAbs'). The derivation is of the form

$$\frac{\Phi\Gamma \cup \{x : \tau'\} \vdash_P e : \tau}{\Phi\Gamma \vdash^J_P \lambda x :: \tau' \to e : \tau' \to \tau} \text{ TypAbs'}$$

Since $\emptyset \vdash_P \Gamma' \cup \{x : \tau'\} \leq \Gamma \cup \{x : \tau\}$, we have by the induction hypothesis that

$$\begin{array}{l} \emptyset, \Gamma' \cup \{x : \tau'\} \models_P e : \rho \text{ succeeds} \\ \emptyset \vdash_P \rho \leq \tau \\ fv(\rho) = \emptyset \end{array}$$

Hence $\emptyset, \Gamma' \models_P \lambda x :: \tau' \to e : \tau' \to \rho$ also succeeds, and we have $\emptyset \vdash_P \tau' \to \rho \leq \tau' \leq \tau$ by SubDepth, as well as $fv(\tau' \to \rho) = \emptyset$.

Case (TypLet). The derivation is of the form

$$\frac{\Gamma \vdash^J_P e : \tau \quad \Gamma \cup \{x : \tau\} \vdash_P e' : \tau'}{\Gamma \vdash^J_P \textbf{let } x = e \textbf{ in } e' : \tau'} \text{ TypLet}$$

By the induction hypothesis we have that

$$\begin{array}{l} \emptyset, \Gamma' \models_P e : \rho \text{ succeeds} \\ \emptyset \vdash_P \rho \leq \tau \\ fv(\rho) = \emptyset \end{array}$$

and since $\emptyset \vdash \Gamma' \cup \{x : \rho\} \leq \Gamma \cup \{x : \tau\}$, we have (again by the induction hypothesis) that $\emptyset, \Gamma' \cup \{x : \rho\} \vdash_P e' : \rho'$ succeeds with the desired properties for ρ'. By CTriv we also have $[\,] \models_P \emptyset$, hence $\emptyset, \Gamma \models_P \textbf{let } x = e \textbf{ in } e' : \rho_1$ directly succeeds, again with the required ρ_1.

Case (TypSub). The derivation is of the form

$$\frac{\Gamma \vdash^J_P e : \tau \quad \emptyset \vdash_P \tau \leq \tau'}{\Gamma \vdash^J_P e : \tau'} \text{ TypSub}$$

By the induction hypothesis we have

$$\begin{array}{l} \emptyset, \Gamma' \models_P e : \rho \text{ succeeds} \\ \emptyset \vdash_P \rho \leq \tau \\ fv(\rho) = \emptyset \end{array}$$

and by SubTrans it follows that $\emptyset \vdash_P \rho \leq \tau'$.

Case (TypStruct). The derivation is of the form

$$\frac{\Pi_{\{l_i\}^i} = \{l_i : \tau \to \tau'_i\}^i \quad \Gamma \vdash^J_P e_i : \tau'_i}{\Gamma \vdash^J_P \{l_i = e_i\}^i : \tau} \text{ TypStruct}$$

By the induction hypothesis we have that

$$\begin{array}{l} \emptyset, \Gamma' \vdash_P e_i : \rho_i \text{ succeeds} \\ \emptyset \vdash_P \rho_i \leq \tau'_i \\ fv(\rho_i) = \emptyset \end{array}$$

for all i. By assumption $fv(\tau, \tau'_i) = \emptyset$, hence $[\,] \models_P \{\rho_i \leq \tau'_i\}^i$ succeeds according to lemma E.5.8. We thus have that $\emptyset, \Gamma' \models_P \{l_i = e_i\}^i : \tau$ succeeds, and τ clearly meets its goals.

Case (TypAlts). The derivation is of the form

$$\frac{\Sigma_{\{k_i\}^i} = \{k_i : \tau'_i \to \tau\}^i \quad \Gamma \vdash^J_P e_i : \tau'_i \to \tau'}{\Gamma \vdash^J_P \{k_i \to e_i\}^i : \tau \to \tau'} \text{ TypAlts}$$

By the induction hypothesis we have that

$$\begin{array}{l} \emptyset, \Gamma' \models_P e_i : \rho_i \\ \emptyset \vdash_P \rho_i \leq \tau'_i \to \tau' \\ fv(\rho_i) = \emptyset \end{array}$$

for all i. It follows from lemma E.5.7 that $\emptyset \vdash_P \rho_i \leq \tau'_i \to \tau'$ must imply $\rho_i = \rho'_i \to \rho''_i$ for some ρ'_i and ρ''_i.

Since $[\tau'/\nu]$ is a solution to $\{\rho''_i \leq \nu\}^i$, and $fv(\rho''_i) = \emptyset$ for all i, we have by lemma E.5.9 a derivation of $\Phi \models_P \{\rho''_i \leq \nu\}^i$, where $\emptyset \vdash_P \Phi\nu \leq \tau'$. By lemma E.5.8 we also have $\Phi \models_P \{\rho_i \leq \tau'_i \to \nu\}^i$. Hence we know that $\emptyset, \Gamma' \models_P$

$\{k_i \to e_i\}^i : \tau \to \Phi\nu$ succeeds, and by SubDepth we know that $\emptyset \vdash_P \tau \to \Phi\nu \leq \tau \to \tau'$. Moreover, since SubHyp is cannot be used in this derivation, P only relates nullary type constructors, and $fv(\tau \to \tau') = \emptyset$, we can conclude that $fv(\tau \to \Phi\nu) = \emptyset$ as well.

The remaining cases are trivial. □

E.5 Various lemmas

Lemma E.5.1. *If $\Phi \models_P C$ and $\alpha \in vars(\Phi)$ then either $\alpha \in fv(C)$, or α is a "new" variable, distinct from any variable in the context of $\Phi \models_P C$.*

Proof. By inspection of the definition in figure 5.6. The only clauses that add variables to the resulting substitution are CVar, where α and β clearly belong to the given constraint set, and CMerge, where α is in the constraint set and the $\overline{\nu}$ are new by definition. □

Lemma E.5.2. *If $C, \Gamma \vdash_P e : \sigma$ then $C \cup C', \Gamma \vdash_P e : \sigma$.*

Lemma E.5.3. *If $C \vdash_P D$ then $C \cup C' \vdash_P D$.*

Lemma E.5.4. *Assume for some fixed α that $\alpha \in \tau^+$ implies $C \vdash \rho \leq \rho'$, and $\alpha \in \tau^-$ implies $C \vdash \rho' \leq \rho$. Then $C \vdash_P [\rho/\alpha]\tau \leq [\rho'/\alpha]\tau$.*

Proof. By structural induction on τ.

Case (β). There are two alternatives:

1. If $\beta = \alpha$ then $C \vdash_P \rho \leq \rho'$ holds by assumption, since $\alpha \in \alpha^+$.
2. If $\beta \neq \alpha$ then we can apply rule SubRefl to obtain $C \vdash_P \beta \leq \beta$ as required.

Case ($t\ \tau_1 \ldots \tau_{n_t}$). By the induction hypothesis, $C \vdash_P [\rho/\alpha]\tau_i \leq [\rho'/\alpha]\tau_i$ holds for each $i \in t^+$. Furthermore, since for an $i \in t^-$, $\alpha \in \tau_i^+$ implies $\alpha \in (t\ \tau_1 \ldots \tau_{n_t})^-$, and $\alpha \in \tau_i^-$ implies $\alpha \in (t\ \tau_1 \ldots \tau_{n_t})^+$, we also have $C \vdash_P [\rho'/\alpha]\tau_i \leq [\rho/\alpha]\tau_i$ for each $i \in t^-$ by the induction hypothesis. Using rule SubDepth we can now derive $C \vdash [\rho/\alpha](t\ \tau_1 \ldots \tau_{n_t}) \leq [\rho'/\alpha](t\ \tau_1 \ldots \tau_{n_t})$ as required.

□

Lemma E.5.5. *If C is a Γ-constraint and Φ unifies C then $fv(\Phi C) \subseteq fv(\Phi\Gamma)$.*

Lemma E.5.6. *If $\overline{\beta} \notin fv(\forall\overline{\alpha}.\tau)$ then $\emptyset \vdash_P \forall\overline{\alpha}.\tau \leq \forall\overline{\beta}.[\overline{\rho}/\alpha]\tau$.*

Lemma E.5.7. *If $P = \{t_i < s_i\}^i$ then $(\emptyset, P)$ is unambiguous.*

Lemma E.5.8. *If $\Phi \models_P C$, $\emptyset \vdash_P C'$, and $fv(C') = \emptyset$, then $\Phi \models_P C \cup C'$.*

Lemma E.5.9. *Let $C = \{\tau_i \leq \alpha_i\}^i \cup \{\alpha_j \leq \rho_j\}^j$ where $fv(\tau_i, \rho_j) = \emptyset$, and let P conform to the requirements in definition 5.3.14. Furthermore, let $\{\tau \leq \alpha, \alpha \leq \rho\} \subseteq C$ imply that $\tau = \rho$. Then:*

If $\emptyset \vdash_P \Phi C$ then $\Phi' \models_P C$ such that $\emptyset \vdash_P \Phi'\alpha \leq \Phi\alpha$ for all α such that $(\tau \leq \alpha) \in C$, and $\emptyset \vdash_P \Phi\alpha \leq \Phi'\alpha$ for all α such that $(\alpha \leq \rho) \in C$.

Proof. We show how to build a derivation of $\Phi' \models_P C$ by induction on the size of C (as defined in the proof of proposition 5.3.4).

If C is empty we have $\Phi' = [\,]$ by CTriv, and the lemma holds vacuously. Otherwise, C must match $C' \cup \{t_i\ \overline{\tau}_i \leq \alpha\}^i \cup \{\alpha \leq s_i\ \overline{\rho}\}^i$ for some $\alpha \notin fv(C, \overline{\tau}_i, \overline{\rho}_i)$. Let $t\ \overline{\tau} = \Phi\alpha$. We now have two cases:

- If all t_i and s_i are equal, we know that $(\sqcup_P \overline{t_i}) \parallel (\sqcap_P \overline{s_j})$ trivially succeeds with t. Now let

$$\begin{array}{rcl} \overline{\nu} & = & \nu_1 \ldots \nu_{n_t} \\ C_l & = & \{\tau_j \leq \nu_j \mid \tau_1 \ldots \tau_{n_t} \in \{\overline{\tau}_i\}^i, j \in t^+\} \cup \\ & & \{\nu_j \leq \tau_j \mid \tau_1 \ldots \tau_{n_t} \in \{\overline{\tau}_i\}^i, j \in t^-\} \\ C_u & = & \{\rho_j \leq \nu_j \mid \rho_1 \ldots \rho_{n_t} \in \{\overline{\rho}_i\}^i, j \in t^+\} \cup \\ & & \{\nu_j \leq \rho_j \mid \rho_1 \ldots \rho_{n_t} \in \{\overline{\rho}_i\}^i, j \in t^-\} \\ C'' & = & C' \cup C_l \cup C_u \end{array}$$

  If C_u is empty, then C'' clearly meets the preconditions of the current lemma. The same holds if C_l is empty. But if both C_l and C_u are nonempty, this means that α has lower as well as upper bounds in C, and then the assumptions about C guarantee that all $\overline{\tau}_i$ and $\overline{\rho}_i$ are equal. Thus, in all three cases we have that C'' meets the preconditions of our lemma.

  Furthermore, we know that Φ solves C, i.e. in particular we have derivations of $\emptyset \vdash_P \{t_i\ \overline{\tau}_i \leq t\ \overline{\tau}\}^i \cup \{t\ \overline{\tau} \leq s_i\ \overline{\rho}_i\}^i$. And by lemma E.5.7 we must also have derivations of the natural premises to the previous judgements — i.e. $\emptyset \vdash_P \Phi[\overline{\tau}/\overline{\nu}]C''$

  The induction hypothesis can now be applied to obtain a derivation of $\Phi' \models_P C''$, where Φ' has the desired properties w.r.t. C'' and $[\overline{\tau}/\overline{\nu}] \circ \Phi$. By repeated use of CDepth we can also construct a derivation of $\Phi' \models_P C' \cup \{t_i\ \overline{\tau}_i \leq t\ \overline{\nu}\}^i \cup \{t\ \overline{\nu} \leq s_i\ \overline{\rho}_i\}^i$, and so by CMerge we have $\Phi' \circ [t\ \overline{\nu}/\alpha] \vdash_P C' \cup \{t_i\ \overline{\tau}_i \leq \alpha\}^i \cup \{\alpha \leq s_i\ \overline{\rho}\}^i$.

  If α has no upper bounds in C, we know that $\emptyset \vdash_P \Phi'\nu_j \leq [\overline{\tau}/\overline{\nu}]\nu_j$ for all $j \in t^+$ (and vice versa for all $j \in t^-$). Hence by SubDepth we have $\emptyset \vdash_P \Phi'[t\ \overline{\nu}/\alpha]\alpha \leq \Phi\alpha$. A corresponding result is obtainable if α has no lower bounds. But if α has both lower and upper bounds, C'' actually expresses equality constraints on $\overline{\nu}$, in which case $\Phi'\overline{\nu}$ must be equal to $\overline{\tau}$. This means that $\Phi'[t\ \overline{\nu}/\alpha]\alpha = \Phi\alpha$, and hence $\Phi' \circ [t\ \overline{\nu}/\alpha]$ meets the requirements of the demanded substitution.

- If all t_i and s_i are *not* equal, then either $\emptyset \vdash_P t_i\ \overline{\tau}_i \leq t\ \overline{\tau}$ or $\emptyset \vdash_P t\ \overline{\tau} \leq s_i\ \overline{\rho}_i$ contains a SubConst step for some t_i or s_i. Since P only relates nullary constructors by assumption, this implies that $\overline{\tau}$, as well as all $\overline{\tau}_i$ and $\overline{\rho}_i$, are empty.

  Furthermore, the assumptions on C now also guarantee that either $\{t_i\}^i$ or $\{s_i\}^i$ is empty. Since we know that t is an upper (lower) bound for $\{t_i\}^i$ ($\{s_i\}^i$), we can deduce from the assumptions on P that $t' = (\sqcup_P \overline{t_i}) \parallel (\sqcap_P \overline{s_j})$ is a lub (glb) for $\{t_i\}^i$ ($\{s_i\}^i$).

  By the induction hypothesis we have a derivation of $\Phi' \models_P C'$ with the desired properties for Φ', and by repeated use of CSub we can build a derivation of $\Phi' \models_P C' \cup \{t_i \leq t'\}^i \cup \{t' \leq s_i\}^i$. By CMerge we obtain $\Phi' \circ [t'/\alpha] \models_P C' \cup \{t_i \leq \alpha\}^i \cup \{\alpha \leq s_i\}^i$. Since we have that C in this case is *either* $C' \cup \{t_i \leq \alpha\}^i$ or $C' \cup \{\alpha \leq s_i\}^i$, we can conclude that $\Phi' \circ [t'/\alpha]$ meets the requirements of the demanded substitution.

□

Bibliography

[AC96] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, New York, 1996.

[Ach96] Peter Achten. *Interactive Functional Programs – Models, Methods, and Implementations*. PhD thesis, Katholieke Universiteit Nijmegen, February 1996.

[AG94] A Arora and MG Gouda. Distributed reset. *IEEE Transactions on Computers*, 43:1026–1038, 1994.

[Agh86] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[AKY90] Y Afek, S Kutten, and M Yung. Memory-efficient self-stabilization on general networks. In *WDAG90 Distributed Algorithms 4th International Workshop Proceedings, Springer-Verlag LNCS:486*, pages 15–28, 1990.

[Ame87] P. America. Inheritance and subtyping in a parallel object-oriented language. In J. Bézivin, J.-M. Hullot, P. Cointe, and H. Lieberman, editors, *Proceedings of ECOOP'87, Paris, France*, Lecture Notes in Computer Science 276, pages 234–242. Springer-Verlag, Berlin, 1987.

[AMST97] G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7:1–72, 1997.

[AO93] Gregory R. Andrews and Ronald A. Olsson. *The SR programming language*. The Benjamin/Cummings Publishing Company, 1993.

[App92] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 92.

[APSV91] B Awerbuch, B Patt-Shamir, and G Varghese. Self-stabilization by local checking and correction. In *FOCS91 Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science*, pages 268–277, 1991.

[Aug87] L. Augustsson. *Compiling Lazy Functional Languages, Part II*. PhD thesis, Department of Computer Science, Chalmers University of Technology, Göteborg, Sweden, November 1987.

[Aug98] Lennart Augustsson. Cayenne – a language with dependent types. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, pages 239–250, September 1998.

[AVWW96] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang*. Prentice Hall, 1996.

[AW93] A. Aiken and E. Wimmers. Type inclusion constraints and type inference. In *ACM Functional Programming and Computer Architecture*, Copenhagen, Denmark, June 1993.

[Bac78] J. Backus. Can Programming be Liberated from the von Neumann Style? A functional style and its algebra of programs. *Communications of the ACM*, 21:280–294, August 1978.

[BB90] G. Berry and G. Boudol. The Chemical Abstract Machine. In *ACM Principles of Programming Languages*, pages 81–94, San Francisco, CA, January 1990.

[BB91] A. Benveniste and G. Berry. The Synchronous Approach to Reactive and Real-time Systems. Technical Report 1445, INRIA-Rennes, 1991.

[BCC+96] Kim B. Bruce, Luca Cardelli, Giuseppe Castagna, the Hopkins Objects Group (Jonathan Eifrig, Scott Smith, Valery Trifonov), Gary T. Leavens, and Benjamin Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1996.

[BDS96] F. Boussinot, G. Doumenc, and J.B. Stefani. Reactive Objects. *Annals of Telecommunications*, 51(9-10):459–473, 1996.

[BG92] G. Berry and G. Gonthier. The Esterel synchronous programming language: design, semantics,implementation. *Science of Computer Programming*, 2(19):87–152, 1992.

[BM97] François Bourdoncle and Stephan Merz. Type-checking higher-order polymorphic multi-methods. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 302–315, Paris, France, 15–17 January 1997.

[BOSW98] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In Craig Chambers, editor, *Object Oriented Programing: Systems, Languages, and Applications (OOPSLA)*, ACM SIGPLAN Notices volume 33 number 10, pages 183–200, Vancouver, BC, October 1998.

[BP89] JE Burns and J Pachl. Uniform self-stabilizing rings. *ACM Transactions on Programming Languages and Systems*, 11:330–344, 1989.

[BS98] F. Boussinot and J-F. Susini. The SugarCubes Tool Box: A Reactive Java Framework. *Software – Practice and Experience*, 28(14):1531–1550, December 1998.

[Bur88] F. W. Burton. Nondeterminism with referential transparency in functional programming languages. *The Computer Journal*, 31(3):243–247, June 1988.

[Car93] Luca Cardelli. An implementation of $F_{\leq}$. Technical Report Research report 97, DEC Systems Research Center, February 1993.

[CH74] R. H. Campbell and A. N. Habermann. The specification of process synchronization by path expressions. In E. Gelenbe and C. Kaiser, editors, *Operating Systems, Proc.*, volume 16 of *LNCS*, pages 89–102. Springer Verlag, April 1974.

[CH93] Magnus Carlsson and Thomas Hallgren. Fudgets – A Graphical User Interface in a Functional Language. In *ACM Functional Programming and Computer Architecture*, Copenhagen, Denmark, June 1993.

[CJJ+98a] Christopher Colby, Lalita Jategaonkar Jagadeesan, Radha Jagadeesan, Konstantin Läufer, and Carlos Puchol. Objects and concurrency in triveni: A telecommunication case study in java. In *Conf. Object-Oriented Technologies and Systems (COOTS '98)*, Santa Fé, Nuevo México, April 1998. USENIX.

[CJJ+98b] Christopher Colby, Lalita Jategaonkar Jagadeesan, Radha Jagadeesan, Konstantin Läufer, and Carlos Puchol. Design and implementation of triveni: A process-algebraic api for threads + events. In *Proc. Intl. Conf. on Computer Languages (ICCL '98)*, Chicago, May 1998. IEEE Computer Society.

[CMV97] K. Claessen, E. Meijer, and T. Vullinghs. Implementing Graphical Paradigms in TkGofer. In *Proceedings of the 1997 International Conference on Functional Programming*, Amsterdam, 1997. ACM.

[Coo91] William R. Cook. Object-oriented programming versus abstract data types. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages, REX School/Workshop, Noordwijkerhout, The Netherlands, May/June 1990*, volume 489 of *Lecture Notes in Computer Science*, pages 151–178. Springer-Verlag, New York, N.Y., 1991.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4), 1985.

[Dij65] E. W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages*, New York, 1965. Academic Press.

[Dij68] E.W. Dijkstra. The Structure of the THE Multiprogramming System. *Communications of the ACM*, 11(5):341–346, 1968.

[Dij74] EW Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the ACM*, 17:643–644, 1974.

[DIM93] S Dolev, A Israeli, and S Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distributed Computing*, 7:3–16, 1993.

[DMN68] Ole-Johan Dahl, Bjorn Myrhaug, and Kristen Nygaard. SIMULA 67. common base language. Technical Report Publ. No. S-2, Norwegian Computing Center, Oslo, Norway, May 1968. Revised Edition: Publication No. S-22.

[Ell97] Conal Elliot. Functional Reactive Animation. In *Proceedings of the 1997 International Conference on Functional Programming*, Amsterdam, 1997. ACM.

[EST95] J. Eifrig, S. Smith, and V. Trifonov. Sound Polymorphic Type Inference for Objects. In *OOPSLA '95*. ACM, 1995.

[FA96] Manuel Fahndrich and Alex Aiken. Making set-constraint program analyses scale. Technical Report CSD-96-917, University of California, Berkeley, September 1996.

[Fil94] A. Filinski. Representing monads. In ACM, editor, *Proceedings of 21st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 446–457, New York, NY, USA, 1994. ACM Press.

[FJ94] Sigbjorn Finne and Simon Peyton Jones. Programming Reactive Systems in Haskell. In *Proceedings of the Glasgow Functional Programming Group Workshop*, Ayr, September 1994. Springer Verlag.

[FJ95] Sigbjorn Finne and Simon Peyton Jones. Composing Haggis. In *Proceedings of the Fifth Eurographics Workshop on Programming Paradigms in Computer Graphics*, Maastrict, Netherlands, September 1995.

[FM89] Y. Fuh and P. Mishra. Polymorphic subtype inference: Closing the theory-practice gap. In *Theory and Practice of Software Development*, Barcelona, Spain, March 1989. Springer Verlag.

[FM90] Y. Fuh and P. Mishra. Type Inference with Subtypes. *Theoretical Computer Science*, 73, 1990.

[Fre94] Tim Freeman. *Refinement Types for ML*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburg, PA, March 1994.

[Gas97] Benedict R. Gaster. Polymorphic Extensible Records for Haskell. In *Proceedings of the Haskell Workshop*, Amsterdam, Holland, 1997.

[Gea98] David Geary. Swing and multithreading. *Java Report: The Source for Java Development*, 3(8), August 1998.

[GJS96] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Java Series. Sun Microsystems, 1996.

[GM91] MG Gouda and N Multari. Stabilizing communication protocols. *IEEE Transactions on Computers*, 40:448–458, 1991.

[GMP89] Alessandro Giacalone, Prateek Mishra, and Sanjiva Prasad. FACILE: A symmetric integration of concurrent and functional programming. *International Journal of Parallel Programming*, 18(2):121–160, April 1989.

[GR85] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, 1985.

[Ham94] K. Hammond. Parallel Functional Programming: An Introduction (Invited Paper). In *PASCO '94*, Linz, Austria, September 1994.

[HC95] Thomas Hallgren and Magnus Carlsson. Programming with Fudgets. In *Advanced Functional Programming*, LNCS 925, pages 137–182. Springer Verlag, 1995.

[Hen82] P. Henderson. Purely Functional Operating Systems. In *Functional Programming and its Applications*, pages 177–189. Cambridge University Press, 1982.

[Hen96] Fritz Henglein. Syntactic properties of polymorphic subtyping. TOPPS Technical Report (D-report series) D-293, DIKU, University of Copenhagen, May 1996.

[HHJW94] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip Wadler. Type classes in Haskell. In Donald Sannella, editor, *Programming Languages and Systems—ESOP'94, 5th European Symposium on Programming*, volume 788 of *Lecture Notes in Computer Science*, pages 241–256, Edinburgh, U.K., 11–13 April 1994. Springer.

[HM95] M. Hoang and J. Mitchell. Lower Bounds on Type Inference With Subtypes. In *ACM Principles of Programming Languages*, San Francisco, CA, January 1995. ACM Press.

[Hoa74] C.A.R. Hoare. Monitors: An Operating Systems Structuring Concept. *Communications of the ACM*, 17(10):549–557, 1974.

[Hol83] Sören Holmström. PFL: A Parallel Functional Language and Its Implementation. Technical Report PMG-7, Programming Methodology Group, Chalmers University of Technology, 1983.

[Hug90] J. Hughes. Why Functional Programming Matters. In D. Turner, editor, *Research Topics in Functional Programming*. Addison Wesley, 1990.

[Hyy87] Kalevi Hyyppä. Optical navigation system using passive identical beacons. In *Proc. Intelligent Autonomous Systems*, Amsterdam, 1987. North-Holland.

[IJ90] A Israeli and M Jalfon. Self-stabilizing ring orientation. In *WDAG90 Distributed Algorithms 4th International Workshop Proceedings, Springer-Verlag LNCS:486*, pages 1–14, 1990.

[Jef95] Alan Jeffrey. A fully abstract semantics for a concurrent functional language with monadic types. In *Proceedings, Tenth Annual IEEE Symposium on Logic in Computer Science*, pages 255–264, San Diego, California, 26–29 June 1995. IEEE Computer Society Press.

[JJM97] S. Peyton Jones, M. Jones, and E. Meijer. Type classes: Exploring the design space. In *Proceedings of the Haskell Workshop*, Amsterdam, Holland, June 1997.

[JM93] L. Jategaonkar and J.C. Mitchell. Type inference with extended pattern matching and subtypes. *Fund. Informaticae*, 19:127–166, 1993.

[Jon93] M.P. Jones. A System of Constructor Classes: Overloading and Implicit Higher-Order Polymorphism. In *ACM Functional Programming and Computer Architecture*, Copenhagen, Denmark, June 1993. ACM Press.

[Jon96a] M.P. Jones. Hugs 1.3, The Haskell User's Gofer System: User Manual. Technical Report NOTTCS-TR-96-2, Department of Computer Science,University of Nottingham, 1996.

[Jon96b] M.P. Jones. Using Parameterized Signatures to Express Modular Structure. In *ACM Principles of Programming Languages*, St Petersburg, FL, January 1996. ACM Press.

[Jon97] M.P. Jones. First-Class Polymorphism with Type Inference. In *ACM Principles of Programming Languages*, Paris, France, January 1997. ACM Press.

[Jon94] M. P. Jones. The implementation of the Gofer functional programming system. Technical Report YALEU/DCS/RR-1030, Department of Computer Science, Yale University, New Haven, Connecticut, USA, May 1994, May 94.

[Kae92] S. Kaes. Type inference in the presence of overloading, subtyping and recursive types. In *ACM Lisp and Functional Programming*, pages 193–204, San Francisco, CA, June 1992.

[KP93] S Katz and KJ Perry. Self-stabilizing extensions for message-passing systems. *Distributed Computing*, 7:17–26, 1993.

[Läu92] K. Läufer. *Polymorphic Type Inference and Abstract Data Types*. PhD thesis, New York University, July 1992.

[Lev83] Daniel Leviant. Polymorphic type inference. In *Conference Record of POPL '83: The 10th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 88–98, 1983.

[Llo95] J. W. Lloyd. Declarative programming in escher. Technical Report CSTR-95-013, Department of Computer Science, University of Bristol, June 1995.

[LPJ95] J. Launchbury and S. Peyton Jones. State in Haskell. *Lisp and Symbolic Computation*, 8(4):293–341, December 1995.

[LPS92] Nancy Lynch and Boaz Patt-Shamir. 6.852 distributed algorithms lecture notes: Self-stabilization. Technical report, Laboratory of Computer Science, MIT, Cambridge, MA, Dec 1992.

[LT89] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, 1989.

[Mac82] B. MacLennan. Values and Objects in Programming Languages. *ACM SIGPLAN Notices*, 17(12):70–80, 1982.

[MBL97] Andrew C. Myers, Joseph A. Bank, and Barbara Liskov. Parameterized types for Java. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 132–145, Paris, France, 15–17 January 1997.

[Mil78] R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[Mil91] R. Milner. The polyadic π-calculus: A tutorial. Technical Report ECS-LFCS-91-180, Lab for Foundations of Computer Science, Edingburgh, October 1991.

[Mit84] J. Mitchell. Coercion and type inference. In *ACM Principles of Programming Languages*, 1984.

[Mit91] J. Mitchell. Type inference with simple subtypes. *Journal of Functional Programming*, 1(3):245–285, 1991.

[MM82] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2), 1982.

[Mog91] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[Mor98] A. K. Moran. *Call-by-name, Call-by-need, and McCarthy's Amb*. PhD thesis, Department of Computing Science, Chalmers University of Technology, Gteborg, Sweden, September 1998.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[MW97] Simon Marlow and Philip Wadler. A practical subtyping system for Erlang. In *ICFP*, pages 136–149, Amsterdam,Holland, June 1997.

[MY93] S. Matsuoka and A. Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 107–150. MIT Press, Cambridge (MA), USA, 1993.

[NC97] Johan Nordlander and Magnus Carlsson. Reactive Objects in a Functional Language – An escape from the evil "I". In *Proceedings of the Haskell Workshop*, Amsterdam, Holland, 1997.

[Nie87] O. M. Nierstrasz. Active objects in hybrid. *ACM SIGPLAN Notices*, 22(12):243–253, December 1987.

[Nor95] Johan Nordlander. Lazy Computations in an Object-oriented Language for Reactive Programming. In *Proc. ACM SIGPLAN Workshop on State in Programming Languages*, San Fransisco, CA, January 1995. Available as Technical Report UIUCDCS-R-95-1900, University of Illinois at Urbana-Champaign.

[Nor98] Johan Nordlander. Pragmatic Subtyping in Polymorphic Languages. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, Baltimore,MD, September 1998.

[Ous94] J. K. Ousterhout. *Tcl and Tk Toolkit.* Addison-Wesley, 1994.

[OW97] Martin Odersky and Philip Wadler. Pizza into Java: Translating Theory into Practice. In *ACM Principles of Programming Languages*, Paris, France, January 1997. ACM Press.

[Pap94] M. Papathomas. *Language Design Rationale and Semantic Framework for Concurrent Object-Oriented Programming.* PhD thesis, Department of Computer Science, University of Geneva, Switzerland, January 1994.

[Pet97] J. Peterson. The Haskell Home Page. `http://haskell.org`, 1997.

[PGF96] S.L. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *ACM Principles of Programming Languages*, pages 295–308, St Petersburg, FL, January 1996. ACM Press.

[PJ87] S.L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, 1987.

[PJ96] S.L. Peyton Jones. Compiling Haskell by program transformation: A report from the trenches. In Hanne Riis Nielson, editor, *Programming Languages and Systems—ESOP'96, 6th European Symposium on Programming*, volume 1058 of *Lecture Notes in Computer Science*, pages 18–44, Linköping, Sweden, 22–24 April 1996. Springer.

[Pot96] Francois Pottier. Simplifying subtyping constraints. In *International Conference on Functional Programming*, page 122 to 133, May 1996.

[Pot98] Francois Pottier. A framework for type inference with subtyping. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, pages 228–238, September 1998.

[PRH+99] Simon Peyton Jones, Alastair Reid, Tony Hoare, Simon Marlow, and Fergus Henderson. A semantics for imprecise exceptions. In *Proc. Programming Languages Design and Implementation (PLDI'99)*, Atlanta, GA, 1999.

[PS97] B. Pierce and M. Steffen. Higher-order subtyping. *Theoretical Computer Science*, 176(1–2):235–282, April 1997.

[PT94] B. Pierce and D.N. Turner. Concurrent Objects in a Process Calculus. In *Theory and Practice of Parallel Programming*, Sendai, Japan, November 1994.

[PT98a] Benjamin C. Pierce and David N. Turner. Local type inference. In *Conference Record of POPL '98: the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1998.

[PT98b] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. mit, 1998.

[Puc98] Riccardo Pucella. Reactive programming in Standard ML. In *Proceedings of the IEEE International Conference on Computer Languages*, pages 48–57. IEEE Computer Society Press, 1998.

[RBP+90] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-oriented Modeling and Design*. Prentice-Hall, 1990.

[Reh97] J. Rehof. Minimal typings in atomic subtyping. In *ACM Principles of Programming Languages*, Paris, France, January 1997. ACM Press.

[Rep92] J. Reppy. Concurrent ML: Design, Application and Semantics. In *Programming, Concurrency, Simulation and Automated Reasoning*, LNCS 693. Springer Verlag, 1992.

[Rey94] J. Reynolds. User Defined Types and Procedural Data Structures as Complementary Approaches to Data Abstraction. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming*, Cambridge, MA, 1994. MIT Press.

[RR96] J. Reppy and J. Riecke. Simple objects for Standard ML. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, May 1996.

[RTL+91] R. K. Raj, E. Tempero, H. M. Levy, A. P. Black, N. C. Hutchinson, and E. Jul. Emerald: A general-purpose programming language. *Software – Practice and Experience*, 21(1):91–118, January 1991.

[RV97] Didier Rémy and Jérôme Vouillon. Objective ML: A simple object-oriented extension of ML. In *ACM Principles of Programming Languages*, Paris, France, January 1997.

[Sar93] J. Sargeant. Uniting Functional and Object-Oriented Programming (invited paper). In *Object Technologies for Advanced Software*, LNCS 742, Kanazawa, Japan, November 1993.

[Sch98] Enno Scholz. Imperative streams – A monadic combinator library for synchronous programming. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, pages 261–272, September 1998.

[Seq98] Dilip Sequeira. *Type Inference with Bounded Quantification*. PhD thesis, University of Edinburgh, 1998.

[SF98] Olin Shivers and Marc Feeley. ICFP Functional Programming Contest. http://www.ai.mit.edu/extra/icfp-contest/, 1998.

[Smi94] G. Smith. Principal type schemes for functional programs with overloading and subtyping. *Science of Computer Programming*, (23):197–226, 1994.

[Sny86] A. Snyder. Encapsulation and Inheritance in Object-oriented Programming Languages. *ACM SIGPLAN Notices*, 21(11):38–45, November 1986.

[Ste97] Martin Steffen. Polarized higher-order subtyping (extended abstract). In Zhaohui Luo and Sergei Soloviev, editors, *Electronic Proceedings of the Types working group Workshop on Subtyping, inheritance and modular development of proofs*, August 1997.

[Sto86] W. Stoye. Message-based Functional Operating Systems. *Science of Computer Programming*, 6:291–311, 1986.

[Str89] Bjarne Stroustrup. Multiple inheritance for C++. *USENIX Computing Systems*, 2(4):367–395, Fall 1989.

[TA88] Anand Tripathi and Mehmet Aksit. Communication, scheduling and resource management in SINA. *Journal of Object-Oriented Programming*, 2(4):24–36, November 1988.

[TS96a] Valery Trifonov and Scott Smith. Subtyping constrained types. In *Proceedings of the Third International Static Analysis Symposium*, volume 1145 of *LNCS*, pages 349–365. SV, September 1996.

[TS96b] Valery Trifonov and Scott Smith. Subtyping Constrained Types. In *Third International Static Analysis Symposum*, LNCS 1145, Aachen, Germany, September 1996.

[Tur81] D. A. Turner. The Semantic Elegance of Applicative Languages. In *Proceedings 1981 Conference on Functional Languages and Computer Architecture*, Wentworth-by-the-Sea, Portsmouth, New Hampshire, 1981.

[Tur87] D. Turner. Functional Programming and Communicating Processes. In *Conference on Parallel Languages and Architectures*, LNCS 259, 1987.

[Var93] G Varghese. Self-stabilization by local checking and correction (Ph.D. thesis). Technical Report MIT/LCS/TR-583, MIT, 1993.

[Wad92] Philip Wadler. The essence of functional programming (invited talk). In *19'th ACM Symposium on Principles of Programming Languages*, Albuquerque, New Mexico, January 1992.

[Wad97] Philip Wadler. How to declare an imperative. *ACM Computing Surveys*, 29(3):240–263, September 1997.

[WB89] P. Wadler and S. Blott. How to make *ad-hoc* polymorphism less *ad hoc*. In *16th ACM Symposium on Principles of Programming Languages*, pages 60–76, 1989.

[Weg87] Peter Wegner. The object-oriented classification paradigm. In Peter Wegner and Bruce Shriver, editors, *Research Directions in Object-Oriented Programming*. The MIT Press, 1987.

[Weg96] Peter Wegner. Interactive software technology. In Allen B. Tucker, editor, *Handbook of Computer Science and Engineering*. CRC Press, 1996.

[Wil90] T. Williams. On Inheritance: What It Means and How To Use It. Technical report, Applications Architecture Group, Microsoft Corp., April 1990.

[WWWK94] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. A Note on Distributed Computing. Technical Report SMLI TR-94-29, Sun Microsystems Laboratories, Inc., November 1994.

[WY88] Takuo Watanabe and Akinori Yonezawa. Reflection in an object-oriented concurrent language. *ACM SIGPLAN Notices*, 23(11):306–315, November 1988.

[YSTH87] A. Yonezawa, E. Shibayama, T. Takada, and Y. Honda. Modelling and programming in ABCL/1. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, Computer Systems Series, pages 55–89. MIT Press, Cambridge, MA, 1987.

[YT87] Y. Yokote and M. Tokoro. Experience and Evolution of Concurrent Smalltalk. *SIGPLAN Notices*, 22(12):406–415, December 1987.