Proceedings of the

Haskell Workshop

June 25, 1995
La Jolla, California

A workshop sponsored by ACM and IFIP.

Yale University Research Report
YALEU/DCS/RR-1075

Proceedings of the

Haskell Workshop

June 25, 1995
La Jolla, California

Paul Hudak, Chair
Yale University

A workshop sponsored by ACM and IFIP.

Yale University Research Report
YALEU/DCS/RR-1075

Foreword

This collection of papers was prepared for presentation at the ACM/IFIP-sponsored *Haskell Workshop* held on June 25, 1995, in La Jolla, California, as part of the larger PLDI/PEPM/FPCA conference extravaganza. The Call for Contributions for the workshop read:

> The functional language Haskell is approaching its 5th birthday. There are now several robust and popular implementations of Haskell, and it has been used in a variety of applications, big and small, academic and industrial. This informal workshop is aimed at discussing the future of Haskell: what have we learned, what should be different, and what is the process for change? The forum will consist of invited talks, presentations of submitted papers, specific proposals for change, and open discussions on the most interesting topics. (Although the workshop will not address implementation techniques per se, implementation issues will be considered when sufficiently influenced by language design.)

Submitted papers were informally reviewed by a collection of anonymous referees, with those in this proceedings chosen for presentation based on quality and relevance to the workshop theme. Despite the review process, all papers should be considered as preliminary descriptions of work in progress, and all copyrights remain exclusively with the authors.

I wish to thank the anonymous referees for their helpful and timely reviews, David Wall and Dennis Volpano for their help with local arrangements, John Williams and Barbara Ryder for their general support, and my assistant Linda Joyce for her help in assembling the proceedings.

Paul Hudak
Chair, Haskell Workshop
June 1995

Haskell Workshop

Four Concurrency Primitives for Haskell

Enno Scholz*

Freie Universität Berlin
Institut für Informatik
Takustr. 9, 14195 Berlin

Email: scholz@inf.fu-berlin.de
Fax: +49-30-83875109

Abstract. A monad for concurrent programming that is suitable for being built into Haskell is presented. The monad consists of only four primitives with a very simple semantics. A number of examples demonstrate that monads encapsulating other, more sophisticated communication paradigms known from concurrent functional languages such as Concurrent ML, Facile, and Erlang can be naturally and systematically constructed from the built-in monad in a purely functional way.

The paper argues that minimizing the number and complexity of the concurrency primitives and maximizing the use of purely functional abstractions in the design of concurrent languages helps to remedy a recurrent dilemma, namely, how to keep the language small and rigorously defined, yet to provide the programmer with all the communication constructs required.

An interleaving implementation of the monad has been built by extending Mark Jones's Gofer environment to handle the concurrency primitives. All programs presented in the course of the paper have been executed using this implementation.

1. Introduction

The topic of concurrency has recently received much attention in programming language research, in general, and in functional programming research, in particular. There seems to be a fundamental conflict between the inherent nondeterminism of concurrent computations and the referential transparency that is considered to be the main virtue of functional languages. Thus, languages like Concurrent ML [Perry 93], Facile [Thomsen et al. 93] and Erlang [Armstrong et al. 93] emerged; these sacrifice referential transparency by allowing purely functional computations and computations with side effects to be arbitrarily interspersed.

A technique for reconciling referential transparency and side effects does exist, however: monads. In Haskell, all I/O operations are encapsulated in a special monad. Since it is well-known that communicating processes are a generalization of programs performing I/O, in principle, there is no hindrance to use the same technique to enhance Haskell with primitives related to concurrency.

However, a look into the world of concurrent languages reveals a bewildering multitude of communication paradigms and primitives, and even with respect to functional concurrent languages, the situation is not much clearer. At first glance it seems that concurrency is still very much a moving target, unsuitable to be introduced to Haskell, which is intended to reflect the consensus of the lazy functional programming community. How could a consensus ever be reached regarding the concurrency primitives to be provided?

This paper makes the point that a set of concurrency primitives might be agreed upon if their number and complexity is designed to be *minimal*. Striving for a minimal concurrent extension of Haskell, we present a calculus of only four concurrency primitives that is sufficient to serve as a base for implementing other, arbitrarily more sophisticated, communication paradigms within Haskell, i.e., in a purely equational way. The main part of the paper is devoted to supporting the claim that, using monads, this can be done naturally and systematically. Complete definitions of an Erlang-style actor paradigm and a paradigm in the style of Concurrent ML are given; moreover, an implementation of an Ada-style paradigm is sketched.

We believe that minimizing the number and complexity of the concurrency primitives while maximizing the part of the

*The author's research is supported by a PhD scholarship from the German Research Council (DFG) under grant Ho 1257/1-2.

concurrent program that is defined purely equationally has beneficial effects in the following areas:

Verification: The fewer and the simpler the primitives, the more of a concurrent program's code is written within the functional language. This means that a maximum of the formal reasoning about its properties can be conducted within the well-understood theory of functional programming, i.e., purely equationally. Only a minimum of the formal reasoning must be conducted within the inherently more complex theory of the underlying concurrency calculus.

Semantics: The minimalist approach presented may help to remedy a recurrent problem in the design of concurrent languages, namely, how to keep the language small, coherent, and well-defined, yet to provide the programmer with all the communication constructs required. In reality, concurrent languages seem to fall into two categories: either they rest on secure theoretical foundations but have limited applicability in practice, or they provide all the mechanisms found to be useful in concurrent programming but are so complex as to be difficult to treat formally. An example of the former case is Occam [Inmos Ltd. 84], which is based on the process calculus CSP [Hoare 85], from which it inherits a large body of theoretical properties. Examples of the latter case are the languages SR [Andrews et al. 88] and Ada [Ichbiah 83]. Obviously, the more useful constructs a language contains, the harder it becomes to give a formal semantics to it. Our approach might be a useful compromise. Although giving a formal semantics to the four primitives presented here is beyond the scope of this paper, it should not be impossible. That done, any other programming paradigm defined equationally on top of these primitives would have a well-founded semantics.

Programming style: Using monads to structure sequential functional programs amounts to splitting the programming process in two phases that can be characterized as follows: first a customized, application-specific programming language (the monad) is built which takes care of the bookkeeping; then, the interesting part of the program is written in the clearest possible way. We believe that this approach is very promising in concurrent programming, too. In many concurrent programs, a substantial part of the application's code is used for mapping the application's communication structure to the language's primitives.[1] This code can be wrapped up by first programming a monad encapsulating a communication paradigm adequate for the application, then solving the problem.

Prototyping: Experiments indicate that functional languages have several advantages over imperative languages for the purpose of prototyping sequential programs [Carlson et al. 93]; it is not clear, though, whether these advantages carry over to the prototyping of concurrent programs. Prototyping of concurrent programs seems to require that the communication paradigm used by the prototype and that provided by the implementation language are similar; it makes little sense, for instance, to prototype an Ada application in a language based on message passing. The two-phase technique mentioned above may be particularly useful in prototyping: First the concurrency paradigm of the implementation language is modelled in the functional language, then a prototype is built using this paradigm.

The paper is organized as follows: The next section addresses some related work. Section 3 explains the notation used. In Section 4, the four primitives are presented. Building them into Haskell in the style of the I/O monad enables concurrent computations to be expressed in a referentially transparent way. An operational semantics of the primitives is given and illustrated by a sample application. Section 5 shows how operations may be added to an existing communication paradigm: The built-in paradigm is enhanced with additional operations for remote function call. Section 6 through 8 demonstrate how one communication paradigm may be built on top of another; monads encapsulating communication paradigms resembling those paradigms which form the basis of the concurrent languages Erlang, Concurrent ML, and Ada are presented. Section 9 discusses the lessons learned.

We have built an interleaving implementation of the concurrency monad presented here by extending Mark Jones's Gofer environment [Jones 94] to handle the concurrency primitives. All programs presented in the course of this paper have been executed using this implementation.

2. Related Work

In [Jones, Hudak 93], the potential of using monads to express concurrent computations is explored. To this end, three operations corresponding to our *newChan*, *send* and *receive* primitives are added to the IO monad. The main difference to our work seems to be that Jones and Hudak aim at avoiding nondeterministic semantics. Therefore, they do not make *fork* a primitive which introduces nondeterminism into the language, but rather define it as an ordinary function within the language which serves as an annotation telling the compiler to try to execute two expressions in parallel. While Hudak and Jones highlight the advantages of their approach by comparing them to lazy stream-processing, our work is more in the tradition of [Karlsson 81] and [Perry 90], who use continuations for introducing nondeterminism and concurrency into pure functional languages without sacrificing concurrency. Here, the main objective is not to speed up functional computations by parallel execution in a context where nondeterminism is considered a necessary evil, but rather to enable concurrent applications which are inherently nondeterministic, for instance operating systems and graphical user interfaces, to be expressed in a functional language. We focus on a question, though, that as yet seems to have received little attention in the literature, namely, how the

[1] In concurrent programming, a saying - originally coined to promote monads in another context - seems to fit, namely, that "programmers often find themselves peering through the underbrush at the interesting code somewhere within." [Hall et al. 92].

choice of the concurrency primitives influences the software engineering of concurrent programs in functional languages.

3. Notation

The notation used in this paper is essentially the functional programming language HASKELL [Hudak et al. 92], however, with the following extensions:

We assume a type system with constructor classes and special syntax for monads as documented and implemented in the functional programming system GOFER, version 2.30 [Jones 94]. The syntax for monads is defined as follows: an expression of type *m a*, where *m* is a monad type constructor, is started by keyword *do* followed by a nonempty list of entries, of which the last must be of type *m a* and is called *tail expression*. The others are called *qualifiers*. The rules for transforming a *do* expression into one using *`bind`* are given in Fig. 1. In case more than one rule matches, the first one applies.

```
do { Pat ← Exp; Rest } ⇒ Exp `bind` \Pat → do { Rest }
do { Exp; Rest }       ⇒ Exp `bind` \_   → do { Rest }
do { let { .. }; Rest } ⇒ let { .. } in     do { Rest }
do { [Exp] }           ⇒ result Exp
do { Exp }             ⇒ Exp
```

Fig 1: Special syntax for monads

Note that, in contrast to a qualifier, a tail expression is not changed by the transformation. Furthermore, the equivalence of *[x]* and *result x* in the list monad is adopted to hold for arbitrary monads in this syntax.

In this syntax, the following expression using primitives from the Haskell 1.3 proposal for monadic I/O

```
getChar `bind` \c →
(case c of 'y' → putChar 'Y'
           'n' → putChar 'N') `bind` \_ →
result True
```

becomes

```
do c ← getChar
   case c of 'y' → putChar 'Y'
             'n' → putChar 'N'
   [True]
```

Moreover, we take the licence to declare type synonyms as instances of constructor classes. In Gofer, this is only possible for type constructors of algebraic data types, which means that every type synonym that is to be used as a monad must be enclosed in an algebraic data type of its own, which leads to a slight cluttering of the code.

To make the paper self-contained, an informal description of functions from the Haskell standard prelude is given where they appear in the example programs. For their definitions, refer to [Hudak et al. 92].

4. The Concurrency Primitives

4.1. Interface

The concurrency primitives that are added to Haskell are represented by functions on a monad *Process*. Their signatures are given in Fig. 2.

```
send    :: Chan a → a → Process ()
receive :: Chan a → Process a
fork    :: Process ()→ Process ()
newChan :: Process (Chan a)
```

Fig. 2: The concurrency primitives

A term of type *Process a* is called a *process term*. It encapsulates a concurrent computation yielding a value of type *a*. In analogy to the *IO* monad, we assume that the compiler will know that process terms must be evaluated according to the operational semantics given in Section 3.3. At any time, an arbitrary number of process terms will be evaluated concurrently. Each of these terms is called a *process*.

In general, monads representing a given communication paradigm are abstract data types with a given set of interface functions (besides *bind* and *result*). In the sequel, we will call these interface functions the paradigm's *instructions*. Other functions on the monad which are defined in terms of the instructions are called *operations*.

Processes communicate by point-to-point, synchronous message-passing on typed, first-class channels.

The operations a process can perform are: send a message on a given channel (*send*) of matching type, receive a message on a channel (*receive*), start another process (*fork*), or create a new channel (*newChan*). A process wanting to send a message on a given channel is halted until a process wanting to receive a message on the same channel is found, and vice versa, such that the sender and the receiver of a message are forced to synchronize.

In case that at one time there are many processes wanting to execute a *send* or a *receive* operation on the same channel, they are paired in a nondeterministic manner that is at the implementation's discretion. It is guaranteed, however, that every message sent is received exactly once by one process. Note that the data type *Chan a* is an abstract data type which has a representation that depends on the language implementation. Besides equality, there is only one other function named *toInt* defined on objects of *Chan a*. *toInt* returns different integer values for different channels.

4.2. Implementation

Since the outcome of a concurrent computation is nondeterministic, the monad *Process* cannot be implemented within Haskell. We describe the operational semantics of the

primitives by a transition relation on sets of process terms.

In Fig. 3 where the primitives' operational semantics is defined, *exp [v := exp']* denotes the expression obtained by substituting every free occurrence of variable *v* in expression *exp* by *exp'*. The operator ⊗ denotes the union of two sets with empty intersection.

Initialization:		
do p	⇒	*{do p }*
		where *p :: Process a, a ≠ ()*
Transition:		
ps ⊗ {do send m ch; p}		
⊗ {do m' ← receive ch; p'}	⇒	*ps ∪ { p, p'[m' :=m]}*
ps ⊗ {do [()]}	⇒	*ps*
ps ⊗ {do fork p'; p}	⇒	*ps ∪ { p', p }*
ps ⊗ {do ch ← newChan; p}	⇒	*ps ∪ { p [ch := ch']}*
		where *ch'* fresh
Successful termination:		
ps ⊗ { do [x] }	⇒	*x*
		where *x :: a, a ≠ ()*

Fig. 3: The concurrency primitives' operational semantics

To start the evaluation of a process term *p*, an initial set having *p* as its only element is created. Repeatedly, from the transition rules given in Fig. 3, one that matches the current set is selected nondeterministically and applied to the current set, yielding a new set. This procedure is repeated until until either the initial process has been reduced to normal form *[x]* (which stands for *result x* in our syntax!), or there is no matching rule. In the former case, the computation's result is *x*, in the latter case, the computation is deadlocked. Note that the initial process has type *Process a* where *a ≠ ()* while other processes created in the course of the computation have type *Process ()*. Moreover, note that there is no explicit termination command: processes terminate implicitly when they return a value.

We have now completed the definition of the concurrency primitives' syntax and operational semantics. Obviously, the underlying communication paradigm is of utmost simplicity.

In the sequel, we show that that the primitives provided are not merely suitable for the construction of serious programs, but can indeed serve as a building-blocks for customized communication mechanisms which are considerably more powerful. Two techniques for constructing new communication mechanisms are studied in this paper: on the one hand, an existing communication paradigm can be extended with additional operations; on the other hand, a completely new communication paradigm may be constructed on top of an existing one. In the remainder of this section and in the next one, additional operations on the *Process* paradigm will be developed; the remaining sections are devoted to the implementation of new paradigms.

4.3. Application

To illustrate the use of the concurrency primitives, we develop a generic concurrent divide-and-conquer operation *divAndConq*. At the core of every divide-and-conquer algorithm there are four domain-specific functions which govern the algorithm's behaviour: *isTrivial* is used to determine whether a problem is trivial, in that case it can be solved by *solve*; otherwise it must be divided into two simpler subproblems (using *divide*), whose solutions are combined using *compose*. These functions may conveniently be abstracted from using type classes.

```
class Problem p    where isTrivial :: p → Bool
                         divide    :: p → (p, p)
class Solution s   where compose   :: s → s → s
class Solvable p s where solve     :: p → s
```

The operation *divAndConq* takes a problem as its parameter, creates a subprocess to calculate the solution, and immediately returns a newly-created channel *sChan*, on which the subprocess will eventually make the solution available. In case the problem is trivial, the subprocess sends the solution on *sChan* and terminates. Otherwise, it divides the problem into two subproblems *p1* and *p2*, solves them using *divAndConq*, receives the subsolutions on channels *sChan1* and *sChan2*, composes them, and terminates after sending the final solution on *sChan*.

```
divAndConq :: (Problem p, Solution s, Solvable p s) ⇒
                                        p → Process (Chan s)
divAndConq p =
  do sChan ← newChan
     fork (
        do if isTrivial p then
              do send sChan (solve p)
           else
              do let (p1,p2) = divide p
                 sChan1 ← divAndConq p1
                 sChan2 ← divAndConq p2
                 s1 ← receive sChan1
                 s2 ← receive sChan2
                 send sChan (compose s1 s2))
     [sChan]
```

Taking lists of integers to be the problem and solution domain, the following instantiation of the classes *Problem*, *Solution* and *Solvable* turns *divAndConq* into the concurrent version of a popular sorting algorithm.

```
instance Problem [Int] where
  isTrivial      = (<= 1) . length
  divide (x:xs)  = if null as then ([x], bs)
                               else (as, x:bs)
                   where
                   as = [ x' | x' ← xs, x' <= x ]
                   bs = [ x' | x' ← xs, x' > x ]
```

```
instance Solution [Int]       where compose = (++)
instance Solvable [Int] [Int] where solve   = id

quickSort :: [Int] → Process [Int]
quickSort p =
  do sChan ← divAndConq p
     receive sChan
```

Note that *length*, *null*, *id*, and *(++)* are defined in the standard prelude. *length* returns the length of a list, *null* tests whether a list is empty, *id* is the identity function, and *(++)* concatenates two lists.

5. Adding Remote Function Call

Given the purely functional definition of an abstract data type (abbrev. ADT), we extend the *Process* monad with operations for defining multiple server processes, each with a unique identity, offering the ADT's operations as remote functions to arbitrary client processes.

5.1. Interface

The interface of the remote function call mechanism consists of two operations *newServer* and *(?)*.

```
newServer :: a → Process (ServerId a)
(?) :: ServerId a → (a → (b,a)) → Process b
```

The function *newServer* takes an object, creates a server for it, and returns a reference to the server. The function *(?)* takes as its parameters a reference *ref* to a server for an object of type *a* and a state transformer *f* on type *a* returning an object of type *b*. The command *ref ? f* causes *f* to be applied to the object managed by the server process, which updates its state accordingly, and sends an object of type *b* back to the client.

5.2. Application

Consider the ADT *Dictionary* which offers dictionary services.

```
createDictionary :: [(String,Int)] →  Dictionary
add      :: String → Int →  Dictionary → ((), Dictionary)
delete   :: String →        Dictionary → ((), Dictionary)
lookUp   :: String →        Dictionary → (Int, Dictionary)
```

Its interface consists of one generator function *createDictionary* and three operations *add*, *delete*, and *lookUp*. (Their definitions are omitted here.) The following process *rfcClient* illustrates the creation and use of a remote function call server for objects of type *Dictionary*.

```
rfcClient :: Process ()
rfcClient =
  do let dict1 = createDictionary [ ("Peter", 10000) ]
         dict2 = createDictionary [ ("Paul", 300),
                                    ("Mary", 850) ]
     richPeople ← newServer dict1
     poorPeople ← newServer dict2
     balance ← poorPeople?lookUp "Mary"
     poorPeople?delete "Mary"
     richPeople?add "Mary" (balance + 2000)
```

Note that the derived process functions coexist with the interface functions of *Process*, i.e., one process may use both, for instance, *(?)* and *send*.

5.3. Implementation

Conceptually, a server for an object of a given type *a* is a process having the object as its state. It receives functions of type $a \rightarrow (b, a)$, applies the function to its state, returns the first element of the resulting pair to the caller and update its state with the second element. The problem is how to represent a reference to the server. A general technique is to identify a process by the channel it listens to. Thus, the straightforward solution is to implement *ServerId a* as follows:

```
type ServerId a = Chan (a → (b, a), Chan b)   -- wrong
```

This would mean that a server understands messages consisting of a state transformer function to be executed and a channel on which to return the result. However, this does not work without existential types, because, for different requests, type *b* is expected to vary. How can type *b* be removed from the definition of type *ServerId a*? The solution is to have the requester transmit a message which combines the application of the state transformer and the command sending back the result into one piece of code:

```
type ServerId a = Chan (a → Process a)        -- ok
```

To issue a request using *(?)*, a process sends such a piece of code to the server which takes the server's state *a* as an argument, applies the state transformer to it, yielding an object *(b, a')*, makes *b* available on a channel that the requester listens to, and returns *a'*.

```
(?) :: ServerId a → (a → (b,a)) → Process b
requestChan ? f =
  do replyChan ← newChan
     send requestChan (\a →  do let (b, a') = f a
                                send replyChan b
                                [a])
     receive replyChan
```

The server repeatedly receives such a piece of code, provides it with its state and gets back its new state when executing it.

```
server :: ServerId a → a → Process ()
server requestChan a =
  do request ← receive requestChan
     a' ← request a
     server requestChan a'
```

newServer a is implemented by forking a server process for *a*, supplying it with a new channel on which to listen, and returning the appropriately typed channel.

```
newServer :: a → Process (ServerId a)
newServer a =
  do requestChan ← newChan
```

```
fork (server requestChan a)
[requestChan]
```

5.4. Remarks

Note that the primitives *newVar*, *readVar*, and *writeVar* used by Launchbury and Peyton Jones to incorporate mutable state variables into Haskell [Launchbury, Peyton Jones 94] can be implemented trivially in terms of *newServer* and *(?)*.

6. Actors

In the previous section, one communication paradigm was enhanced with additional operations. This section illustrates how a completely new communication paradigm, an actor paradigm resembling the one on found in Erlang [Armstrong et al. 93], may be built on top of an existing one, namely, the built-in calculus.

6.1. Interface

The actor monad *ACT* implements the paradigm of point-to-point, unidirectional, asynchronous, buffered message passing with explicit, guarded message receipt using an asymmetric naming scheme. That is, sending a message is non-blocking and requires the sender to know the name of the receiver. An actor wishing to receive a message explicitly issues an instruction which causes its execution to halt until a message from an unspecified sender (whose identity is possibly unknown to the receiver) has arrived. Messages sent to an actor are buffered in an unbounded message queue. The order in which two messages were sent is not necessarily that in which they arrive. Each actor in the system is uniquely identified by a process identifier (abbrev. *PID*).

An expression of type *ACT m a* is called an *actor term* and represents a computation which understands messages of type *m* and returns a result of type *a*. (The distinction between *actors* and actor terms is the same as the distinction between processes and process terms.) These are the operations an actor can perform:

```
receiveACT  :: (m → Bool) → ACT m m
sendACT     :: Pid m' → m' → ACT m ()
selfACT     :: ACT m (Pid m)
forkACT     :: ACT m' () → ACT m (Pid m')
```

sendACT takes the receiving actor's PID and the data item to be transmitted as its parameters. *receiveACT* takes a predicate on messages, the guard, as its parameter; it blocks until a message *m* arrives for which the guard evaluates to *True*, then it returns *m*. *selfACT* take no parameters and immediately returns the current actor's PID. *forkACT* takes the actor term to be evaluated concurrently as its parameter and returns the child actor's PID. Note that PIDs, like channels, are typed, such that no actor can receive a message that it does not understand.

6.2. Application

This example actor returns the value *100*. It illustrates how the use of guards enables an actor to process messages in an order that is independent of their order of arrival. Moreover, it illustrates that the operation *sendACT* is non-blocking.

```
mainACT :: ACT Int Int
mainACT =
  do self ← selfACT
     child ← forkACT (do sendACT self 50
                         sendACT self 150)
     sendACT child "hallo"
     a ← receiveACT (> 100)
     b ← receiveACT (< 100)
     [a - b]
```

Note that the implementation of the actor calculus is completely hidden. In particular, an actor cannot execute operations defined on *Process*.

6.3. Implementation

It is well-known that asynchronous communication can be implemented in terms of synchronous communication by use of buffers. The idea is to implement an actor by a process with additional state, namely, a channel on which to listen for messages, and a message buffer.

In [King, Wadler 92], [Jones, Duponcheel 93], and [Jones 95], techniques for combining monads are discussed. Using constructor classes, a *monad transformer S* can be defined which combines the state monad introduced by [Wadler 92] with an arbitrary other monad *m*:

```
type S st m a = st → m (a, st)

instance Monad m ⇒ Monad (S st m) where
  result a  = \st → [(a, st)])
  bind ma f = \st → do (a, st') ← ma st
                       f a st')
```

Here, *st* is the type of the monad's state.

Only three operations on this monad exist. Either its state can be read using *readS*, or its state can be set, using *writeS*, or an operation of the inner monad may be executed, using *innerS*.

```
readS :: Monad (S st m) ⇒ S st m st
readS = \st → [(st, st)]

writeS :: Monad m ⇒ st → S st m ()
writeS st' = \_ → [((), st')]

innerS :: Monad m ⇒ m a → S st m a
innerS ma = \st → do a ← ma
                     [(a, st)]
```

Now a monad enhancing the monad *Process* with the additional state information described above looks like this:

```
type ACT m  =  S (Pid m, [m]) Process
```

An actor is identified by its PID, which is implemented by the channel the actor listens to.

```
type Pid m = Chan m
```

To receive a message, an actor checks whether its message buffer contains a message for which the guard evaluates to true, in which case this message is removed from the buffer and returned immediately. Otherwise, the actor repeatedly executes *receive* and places the arriving messages in the buffer until a message arrives for which the guard evaluates to true.

```
receiveACT :: (m → Bool) → ACT m m
receiveACT guard =
  do (chan, xs) ← readS
     let (xs', xs'') = span (not . guard) xs
     if null xs'' then
        do let loop =
                 do (chan, xs) ← readS
                    x ← innerS (receive chan)
                    if guard x then
                       do [x]
                    else
                       do writeS (chan, xs ++ [x])
                          loop
           loop
     else
        do writeS (chan, xs' ++ tail xs'')
           [head xs'']
```

Note that *span* is defined in the standard prelude. It takes a predicate and a list as its parameters and splits the list into a left and a right part. The left part is the largest initial part of the list such that the predicate is true for all its elements; the right part is the rest of the list.

To send a message to another actor, the sender forks a process that makes the message available on the channel that the receiver is listening to. This way, the sending actor does not block until the message has been processed by the receiver.

```
sendACT :: Pid m' → m' → ACT m ()
sendACT other m =
  do innerS (fork (send other m))
```

Using *selfACT*, an actor gets to know its own PID.

```
selfACT :: ACT m (Pid m)
selfACT =
  do (chan, _) ← readS
     [chan]
```

When a new actor is forked, it is supplied with a new channel to listen to and an empty message buffer. Its state after termination is discarded.

```
forkACT :: ACT m' () → ACT m (Pid m')
forkACT p =
  do chan ← innerS newChan
     innerS (fork (do p (chan, [])
                      [()]))
     [chan]
```

This is how the first actor is started:

```
initActor :: Actor m a → Process a
initActor actor =
  do chan ← newChan
     (a, _) ← actor (chan, [])
     [a]
```

7. Concurrent ML

In this section, the communication paradigm used in Concurrent ML is implemented on top of the built-in paradigm. Essentially, this means adding an operator for external choice, as found in, e.g., Occam and Facile. However, these languages require the guards of the external choice to be syntactically distinguished from ordinary send or receive commands. This requirement severely reduces the modularity of the resulting programs; this topic is discussed at length in [Reppy 88]. In Concurrent ML, and in our calculus, this restriction does not apply.

In the *CML* paradigm, *CML terms* and *CML processes* are defined analogously to process terms and processes, respectively.

7.1. Interface

The *CML* paradigm has four instructions which have the same semantics as those of the built-in paradigm:

```
sendCML    :: ChanCML a → a → CML ()
receiveCML :: ChanCML a → CML a
forkCML    :: CML () → CML ()
newChanCML :: CML (ChanCML a)
```

However, there is one additional instruction:

```
chooseCML  :: [CML a] → CML a
```

While each ordinary process is only ready to execute one instruction at a time, a *CML* process having the form *chooseCML* $[p_1,..,p_n]$ may be ready to execute more than one instruction at a time, namely, all the first instructions of the $p_1,..,p_n$. The choice which one is actually executed is nondeterministic.

7.2. Application

The string returned by the following CML process *mainCML* is either *"p1: first p2: first"* or *"p1: second p2: second"*.

```
mainCML :: CML String
mainCML =
  do chan1 ← newChanCML
     chan2 ← newChanCML
     chan3 ← newChanCML

     let p1 =
        do alt ← chooseCML [
                 do sendCML chan1 1
                    ["p1: first "],
                 do receiveCML chan2
                    ["p1: second "]
               ]
           sendCML chan3 alt

     let p2 =
        do alt ← chooseCML [
                 do chooseCML [
                       do receiveCML chan1
                          ["p2: first "],
                       do chooseCML [
                             do sendCML chan2 2
                                ["p2: second "],
                             do tid p1
                                ["Can't happen!"]
                          ]
                    ]
               ]
     sendCML chan3 alt
     chooseCML [forkCML p1]
     forkCML p2
     s1 ← receiveCML chan3
     s2 ← receiveCML chan3
     [s1 ++ s2]
```

The process illustrates two points about the paradigm:

1. The nesting of *chooseCML* instruction does not matter: *p2* could be rewritten using only one *chooseCML* instruction.
2. An element of a *chooseCML* instruction is not restricted to an instruction sequence starting with a *send* or a *receive*, but may be an arbitrary functional expression, like *tid p1*, or forkCML p2.

The last point is especially important for enabling abstraction and modularity, which was first recognized by Reppy and motivated him to devise "first-class synchronous actions" [Reppy 88] which form the basis of Concurrent ML [Reppy 93]. In Concurrent ML and in the calculus presented here, one can compose server processes $p_1,..,p_n$ to a new server process writing *chooseCML* $[p_1,..,p_n]$, without knowing on which channels $p_1,..,p_n$ want to communicate, whether they want to send or to receive, or knowing anything at all about their implementation. In fact, the guards of the external choice operator may be computed dynamically. This an advantage over languages like Occam or Facile, where the guards of the external choice operator must be known statically.

7.3. Implementation

The idea of the implementation is to have a global *transaction manager* process which any *CML* process consults before executing an instruction. This is done by sending a *transaction request* to the transaction manager.

```
type TAReq    = (Tid, Chan (Maybe Tid), TAKind)
data TAKind   = Send ChanId (Process ())
              / Receive ChanId
              / Fork (Chan Tid)
              / NewChan
type Tid      = Int
data Maybe a  = Yes a
              / No
```

A process issuing a transaction request tags it with its *current transaction identifier* (abbrev. TID). The transaction manager can either *commit*, or *abort*, or *suspend* a transaction request. For every TID, it will commit at most one transaction request tagged with this TID and abort all the others. The *chooseCML* instruction is implemented by forking one *CML* process for each of its element terms. While a process's current TID is usually unique, all processes created to evaluate an element term of a *chooseCML* instruction have the same current TID. In order to execute their first instruction, they all issue transaction requests tagged with the same TID. Eventually, only one of them will be commited, which means it can proceed; all others will be aborted, which means they terminate.

The transaction manager issues the TIDs to the processes; whenever one transaction request is committed, the requesting process is handed a fresh TID. The transaction manager keeps track of the set of valid TIDs. For each transaction request that commits, the TID of the request is removed from the set of valid TIDs and the fresh TID which was handed to the requesting process is added to the set. Moreover, the transaction manager stores all suspended transaction requests, i.e., *Send* or *Receive* requests with no matching request

The transaction manager can be in one of three states: its *initial* state, the *commit* state, or the *purge* state. In the initial state, the transaction manager waits for a transaction request. If the request's TID is not valid at all, i.e., if a transaction request with this TID has committed already, the transaction is aborted straightaway and the transaction manager returns to its initial state. In case the TID is valid, the transaction manager must decide whether to commit the transaction or suspend it. If the request is neither for a *Fork* nor a *NewChan* transaction, it is committed immediately; the transaction manager goes into state *commit*. If the request is for a *Send* or a *Receive* transaction, however, it can only commit together with a matching request. Two transaction match, if one of

them is a *Receive* transaction, the other is a *Send* transaction, they both want to communicate on the same channel, and they don't belong to the same process.

```
matches :: TAReq → TAReq → Bool
matches (tid, _, Send ch) (tid', _, Receive ch') =
    (ch == ch') && (tid /= tid')
matches (tid, _, Receive ch) (tid', _, Send ch') =
    (ch == ch') && (tid /= tid')
matches _ _ =
    False
```

This is the transaction manager's code:

```
taManager :: Chan TAReq → [TAReq]
                        → [Tid] → Tid → Process ()
taManager inChan suspReqs validTids nextTid =
  do (req @ (tid, replyChan, kind)) ← receive inChan
     if not (tid `elem` validTids) then
        do abort req
           taManager inChan suspReqs validTids nextTid
     else
        do case kind of
             Fork _ →
               commit [req] suspReqs
             NewChan  →
               commit [req] suspReqs
             _ →
               case span (not . matches req) suspReqs of
                 (_, [])→
                   taManager inChan (req:suspReqs)
                                          validTids nextTid
                 (reqs1, req':reqs2) →
                   commit (sort [req,req'])(reqs1 ++ reqs2)

  where
  commit [(tid, replyChan, Fork replyChan')] suspReqs' =
     do send replyChan (Yes nextTid)
        send replyChan' (nextTid + 1)
        purge [tid] suspReqs' 2
  commit [(tid, replyChan, NewChan)] suspReqs' =
     do send replyChan (Yes nextTid)
        purge [tid] suspReqs' 1
  commit [(tid, replyChan, Receive _),
          (tid', replyChan', Send _ sendCom)] suspReqs' =
     do send replyChan (Yes nextTid)
        send replyChan' (Yes (nextTid + 1))
     purge [tid,tid'] suspReqs' 2

  purge commitIds suspReqs' idsUsed =
     do let stillOk (tid, _, _) = not (tid `elem` commitIds)
        for (filter (not . stillOk) suspReqs') abort
        taManager inChan
                  (filter stillOk suspReqs')
                  ((validTids \\ commitIds) ++
                          [nextTid..nextTid+idsUsed-1])
                  (nextTid + idsUsed)
```

Note that *for* is a generic function on monads which is defined as follows:

```
for ::[a] → (a → M b) → M [b]
for [] f      = [[]]
for (a:as) f = do b ← f a; bs ← for as f; [b:bs]
```

If there is already a matching request in the transaction manager's list of suspended requests then both the incoming and the matching requests are committed simultaneously. Otherwise, the incoming request is suspended.

Committing one or two transaction requests in state *commit* always involves sending a fresh TID to the requester. In the case of a *Fork* request, an additional fresh PID for initializing the child process must be sent to the requester. After that, those suspended transaction requests that have the same TID as the one just committed are aborted and removed from the store; this is done in state *purge*.

```
abort :: TAReq → Process ()
abort (_, replyChan, _) = send replyChan No
```

A *CML* process term is implemented by an ordinary process term with extra state, namely, its current TID and the channel on which the transaction manager receives requests. Moreover, it it needs an error exit to allow for the possibility of its current transaction being aborted.

```
type CML = S (Tid, Chan TAReq) (M Process)
```

The extra state is provided by the type constructor *S* which was introduced in the previous section. The type constructor *M* (cf. [King, Wadler 92]) adds an error exit to an arbitrary monad *m* by combining it with the *Maybe* monad:

```
type M m a = m (Maybe a)

instance Monad m ⇒ Monad (M m) where
  result a   =  [Yes a]
  bind ma f =  do maybe ← ma
                  case maybe of
                    Yes a → f a
                    No    → No
```

Only two operations are defined on *M*: *exitM* exits from the current computation; *innerM* executes an operation of the inner monad.

```
exitM :: Monad m ⇒ M m ()
exitM s = [No]

innerM :: Monad m ⇒ m a → M m a
innerM ma =  do a ← ma
                [Yes a]
```

Note that *CML* is a doubly nested monad; thus, instructions of the innermost monad *Process* are prefixed with *innerSM*, instructions of the monad *M Process* are prefixed with *innerS*, and instructions of the monad *S st (M Process)* can be accessed without a prefix.

```
innerSM :: Monad m ⇒ m a → S st (M m) a
innerSM = innerS . innerM
```

A process wanting to conduct a transaction of kind *kind* executes instruction *tryTA kind*. A message composed of the process's current TID, a reply channel, and *kind* is then sent to the transaction manager. In case the reply from the transaction manager is negative, the process fails itself. If the reply is positive, it contains a the process's new TID which is to be used for executing the next instruction.

```
tryTA :: TAKind → CML ()
tryTA kind =
  do (tid, tm) ← readS
     replyChan ← innerSM newChan
     innerSM (send tm (tid, replyChan, kind))
     reply ← innerSM (receive replyChan)
     case reply of
       Yes tid' →
         do writeS (tid', tm)
       No →
         do innerS exitM
```

The transaction request for *sendCML* and *receiveCML* contain an integer value encoding the channel on which a value is to be communicated.

```
sendCML :: ChanCML a → a → CML ()
sendCML chan a =
  do tryTA (Send (toInt chan))
     innerSM (send chan a)

receiveCML :: ChanCML a → CML a
receiveCML chan =
  do tryTA (Receive (toInt chan))
     innerSM (receive chan)
```

To fork a new *CML* process, it is initialized with a fresh TID and the channel that the transaction manager is listening to. The resulting state is discarded.

```
forkCML :: CML () → CML ()
forkCML code =
  do replyChan ← innerSM newChan
     tryTA (Fork replyChan)
     tid ← innerSM (receive replyChan)
     (_, tm) ← readS
     innerSM (fork (do Yes ((), _) ← code (tid, tm)
                       [()])
```

A CML channel is implemented by an ordinary channel.

```
type ChanCML = Chan
```

Channel creation in the *CML* paradigm is the same as in the built-in paradigm

```
newChanCML :: CML (ChanCML a)
newChanCML =
  do tryTA NewChan
     innerSM newChan
```

For each of its element terms, *chooseCML* forks one process, which executes the element process and sends the result and the current TID at the time of termination termination on a common channel *aChan*. Since the first transaction of all but one process will be aborted, only one value will ever become available on *aChan*.

```
chooseCML :: [CML a] → CML a
chooseCML ps =
  do (tid, tm) ← readS
     aChan ← innerSM newChan
     for ps (\p →
       do innerSM (fork (
            do res ← p (tid, tm)
               case res of
                 Yes (a, (tid', _)) → send aChan (a, tid')
                 No                 → [()])))
     (a, tid') ← innerSM (receive aChan)
     writeS (tid', tm)
     [a]
```

This is how the initial CML process is started: First, the transaction manager is started in a state where *1* is the only valid TID and 2 is the next TID to be issued. Then, the initial *CML* process is started with an initial TID of *1*.

```
initCML :: CML a → Process a
initCML p =
  do chan ← newChan
     fork (taManager chan [] [1] 2)
     Yes (a, _) ← p (1,chan)
     [a]
```

8. Ada

The last paradigm presented in this paper is an attempt to capture some of the essentials of concurrent tasks in Ada. Tasks communicate by accepting and requesting service rendezvous from each other on the basis of an asymmetric naming scheme where the requesting task must specify the desired server, while a task wanting to accept a service request cannot decide which task it is ready to serve.

8.1. Interface

These are the operations that a process may perform:

```
newService   :: ADA (Service x y)
selfADA      :: ADA TaskId
forkADA      :: ADA () → ADA TaskId
requestADA   :: TaskId → Service x y → x → ADA y
acceptADA    :: [Alt] → ADA ()
```

newService creates a new service reference, i.e., a typed object which is used to tag and match service offers and service requests. *selfADA* and *forkADA* have functionalities analogous to *selfACT* and *forkACT*. *requestADA* takes a server's ID, an operation identifier and the operation's argument as ist parameters. *requestADA* blocks until the chosen server has accepted and executed the operation, then it returns the operation's result. *acceptADA* takes a list of

alternatives as its argument. Each alternative associates one operation identifier with a piece of code handling this operation. An alternative is constructed using operator *(>>)*.

```
(>>) :: Service x y → (x → ADA y) → Alt
```

8.2. Application

The program *mainADA* uses a server task *calculator* whose exceedingly weird behaviour can only be justified with its intended use of illustrating the communication paradigm.

```
mainADA :: ADA Int
mainADA =
  do mult ← newService
     square ← newService

     let calculator :: ADA ()
       calculator =
           do acceptADA [
                 mult >> \ (m, n) →
                       do [m * n]
              ]
              acceptADA [
                 mult >> \pair →
                       do helper ← forkADA calculator
                          requestADA helper mult pair,
                 square >> \n →
                       do acceptADA [
                             square >> \n' →
                                   do [n'*n']
                          ]
                          [n*n]
              ]
           calculator

  server ← forkADA calculator
  a ← requestADA server mult (4,5)
  forkADA (requestADA server square 0)
  requestADA server square a
```

Initially, a *calculator* task offers only one kind of service to its surroundings, namely, the operation *mult*. Having successfully executed it, it offers the operations *mult* and *square*. Should the surroundings choose the operation *mult* to be executed, the calculator starts another calculator and lets it perform the actual multiplication. Should a client task choose the *square* service, it will be blocked until a second client asks for the square service, too; then, both clients will be served and the server will return into its initial state.

Note that this paradigm does not allow for postprocessing to be done, i.e., computations being performed which depend on the service accepted, but taking place after the service's result has been transmitted to the caller. This can be easily remedied by changing the signatures of *acceptADA* and *(>>)* to become

```
acceptADA ::[Alt a] → ADA a
(>>)      :: Service x y → (x → ADA (y, ADA a)) → Alt a
```

8.3. Implementation

Since the implementation of the ADA paradigm introduces no new techniques, it is omitted. Each *Service* object is implemented by a process which matches service requests and service offers in a manner that is similar to, albeit simpler than, that of the transaction manager. The difference is that the decision which transactions to commit and which to abort need not be made by one global authority but can be made by the task executing *acceptADA*, since a *requestADA* statement is not allowed to contain alternatives. The code required for implementing the *ADA* paradigm is slightly more than half the size of the *CML* paradigm's code.

9. Conclusion

The aim of the work that has been presented in this paper is to design a *minimal concurrent extension* for Haskell and to assess its usefulness. To this end, four very simple concurrency primitives were built into Haskell using the technique known from monadic I/O. Apart from assuming language support for monads, as found in Gofer, no changes were made to Haskell; in particular, no new language features related to concurrency were introduced.

To us, the preliminary conclusions of our work are the following:

- Neither laziness, nor static typing, nor referential transparency need be sacrificed on the altar of concurrency.
- In terms of expressive power and readability, programs written using current Haskell technology can compete with programs written in concurrent functional languages like Facile, Erlang and Concurrent ML.

Moreover, it seems that the two-phase approach to programming encouraged by monads has some particularly worthwhile applications in concurrent programming.

References

[**Andrews et al. 88**] G.R. Andrews et al.: *An Overview of the SR Language and Implementation*, ACM Transactions on Programming Languages and Systems, Vol. 10, No.1, 1988

[**Armstrong et al. 93**] J. Armstrong et al.: *Concurrent Programming in Erlang*, Prentice-Hall, 1993

[**Carlson et al. 93**] W.E. Carlson, P. Hudak, M.P. Jones: *An Experiment Using Haskell to Prototype "Geometric Region Servers" for Navy Command and Control*, Research Report 1031, Dept. of Computer Science, Yale University, 1993

[**Hall et al. 92**] C. Hall et al.: *The Glasgow Haskell Compiler: A Retrospective*, 1992 Glasgow Workshop on Functional Programming, Ayr, 1992

[**Hoare 85**] C.A.R. Hoare: *Communicating Sequential Processes*, Prentice-Hall, 1985

[**Hudak et al. 92**] P. Hudak, S. Peyton Jones, P. Wadler (editors): *Report on the Programming Language Haskell: Version 1.1*, ACM SIGPLAN Notices, 27 (5), May 1992

[**Ichbiah 83**] J. Ichbiah (ed.): *Ada Programming Language*, ANSI-MIL-STD-1815A, Ada Joint Program Office, Department of Defense, Washington DC, 1983

[**Inmos Ltd. 84**] Inmos Ltd.: *Occam Programming Manual*, Prentice-Hall, 1984

[**Jones, Duponcheel 93**] M. Jones, L. Duponcheel: *Composing Monads*, Research Report YALEU/DCS/RR-1004, Yale University, 1993

[**Jones 94**] M. Jones: *Gofer 2.21/2.28/2.30 Release Notes*, available by anonymous ftp from `ftp.cs.yale.edu`

[**Jones 95**] M. Jones: *Functional Programming with Overloading and Higher-Order Polymorphism*, First International Spring School on Advanced Functional Programming Techniques, LNCS 925, Springer Verlag, 1995

[**Karlsson 81**] K. Karlsson: *Nebula, a Functional Operating System*, Chalmers University, Göteborg, 1981

[**King, Wadler 92**] D.J. King, P. Wadler: *Combining Monads*, 1992 Glasgow Workshop on Functional Programming, Ayr, 1992

[**Reppy 88**] J.H. Reppy: *Synchronous Operations as First-Class Values*, ACM SIGPLAN Conference on Programming Language Design and Implementation, 1988

[**Perry 90**] N. Perry: *The Implementation of Practical Functional Programming Languages*, PhD thesis, Imperial College, University of London, 1990

[**Reppy 93**] J.H. Reppy: *Concurrent Programming with Events: The Concurrent ML Manual*, AT & T Bell Laboratories, 1993

[**Thomsen et al. 93**] B. Thomsen et al: *Facile Antigua Release Programming Guide*, Technical Report ECRC-93-20, 1993 European Computer-Industry Research Centre

[**Wadler 92**] P. Wadler: *The Essence of Functional Programming*, 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1992

Concurrent Haskell: preliminary version

Simon Peyton Jones
Sigbjorn Finne
Department of Computing Science, University of Glasgow G12 8QQ
simonpj@dcs.glasgow.ac.uk

June 2, 1995

Abstract

Some applications are most easily expressed in a programming language that supports concurrency, notably interactive and distributed systems. We propose extensions to the purely-functional language Haskell that allows it to express explicitly concurrent applications; we call the resulting language Concurrent Haskell.

The resulting system appears to be both expressive and efficient, and we give a number of examples of useful abstractions that can be built from our primitives.

We have developed a freely-available implementation of Concurrent Haskell, and are now using it as a substrate for a graphical user interface toolkit.

1 Introduction

In the `xmh` mail tool, the `Compose` button puts up a fresh window in which a new message can be typed, and then sent. Even before it has been sent, though, the `Compose` button can be pressed again, and a second message composed and sent.

The easiest way to express this interaction when implementing the mail tool uses *concurrency*: each press of the `Compose` button spawns a concurrent process that is responsible for dealing with that one message, and nothing else. In general, graphical user interfaces provide a powerful motivation for supporting concurrency in a programming language.

In this paper we describe a concurrent extension to the functional language Haskell. Our principal motivation is to provide a more expressive substrate upon which to build sophisticated I/O-performing programs. Our earlier work showed how to use monads to express I/O (Peyton Jones & Wadler [1993]), and how the same idea could be generalised to accommodate securely encapsulated mutable state (Launchbury & Peyton Jones [1996]; Launchbury & Peyton Jones [1994]). Concurrent Haskell represents the next step in this research programme, which aims to build a bridge between the tidy world of purely functional programming and the gory mess of of I/O intensive programs.

2 Setting the scene

Before proceeding, it is worth being clear about what this paper is about, and what it is not about.

First, we make a sharp distinction between *(implicit) parallelism* and *(explicit) concurrency*. The goal of implicit parallelism is to increase performance by employing multiple processors. For example, it allows the expression e_1+e_2 to be evaluated in parallel, by evaluating e_1 simultaneously

with e_2. Since Haskell has no side effects, the relative interleaving of their evaluation can have no effect on the result. Furthermore, the runtime system may legitimately choose to evaluate the expressions in sequence, instead of in parallel, if for example all the processors are busy.

The goal of our concurrent extension is quite different: we want to initiate concurrent input-output-performing processes. Process initiation is completely explicit, and the relative rate of execution may affect the overall I/O behaviour (which constitutes the "result"). For example, in the mail-tool example the relative order of arrival of the two electronic mail messages will certainly be influenced by the rate at which their managing processes execute after their **Send** button is pressed.

Second, we are interested in concurrency as a substrate for interaction, and not as a model computation. We have certainly gained useful insights from the design of process calculi such as the π-calculus (Milner, Parrow & Walker [1992]), and (more particularly) the programming language Pict (Pierce & Turner [1995]), but we have no ambition to encode everything as a process.

The closest comparison is with Reppy's Concurrent ML, whose motivations were very similar to ours (Reppy [1992]; Reppy [1991]). Our design has emerged as somewhat different to his, as we will discuss later. The main reasons is, of course, that ML is strict, and I/O is done as a side-effect of expression evaluation, while Haskell is non-strict, and I/O is done through a monad.

An important design principle was to choose primitives that can be implemented as simply and efficiently as possible — even if they therefore indeed are somewhat primitive. This contrasts with a design approach that attempt to provide as primitives the ideal abstractions from the programmer's point of view. We take this approach for two reasons. Firstly, the "ideal abstractions" may differ from program to program or from programmer to programmer, so it seems better to provide the "raw iron" from which these abstractions can be built, rather than to dictate what they should be. Secondly, it is always possible to build nice abstractions on top of efficient primitives (provided the set of primitives is rich enough), but it is *not* possible to build efficient abstractions on top of inefficient primitives, however nice.

3 The basic ideas

Concurrent Haskell adds two main new ingredients to Haskell:

- processes, and a mechanism for process initiation (Section 3.2); and
- atomically-mutable state, to support inter-process communication and cooperation (Section 3.3).

Before we discuss either of these, though, it is necessary to review the monadic approach to I/O introduced by Peyton Jones & Wadler [1993], and adopted by the Haskell language in Haskell 1.3.

The semantics of Concurrent Haskell is discussed later, in Section 7.

3.1 A review of monadic I/O

In a non-strict language it is completely impractical to perform input/output using side-effecting "functions", because the order in which sub-expressions are evaluated — and indeed whether they are evaluated at all — is determined by the context in which the result of the expression is used, and hence is hard to predict. This difficulty can be addressed by treating an I/O-performing computation as a *state transformer*; that is, a function that transforms the current state of the

world to a new state. In addition, we need the ability for an I/O-performing computation to return a result. This reasoning leads to the following type definition:

```
type IO a = World -> (a, World)
```

That is, a value of type `IO t` takes a world state as input, and delivers a modified world state together with a value of type `t`. Of course, the implementation performs the I/O right away — thereby modifying the state of the world "in place".

We call a value of type `IO t` an *action*. Here are two useful actions:

```
getChar :: IO Char
putChar :: Char -> IO ()
```

The action `getChar` reads a character from the standard input, and returns it as the result of the action. `putChar` takes a character and returns an action that writes the character to the standard output.

Actions can be combined in sequence using the infix combinators >> and >>=:

```
>>  :: IO a -> IO b -> IO b
>>= :: IO a -> (a -> IO b) -> IO b
```

For example, an action that reads a character and then prints it twice is [1]:

```
getChar       >>= \c ->
putChar c     >>
putChar c
```

The sequencing combinators, >> and >>=, feed the result state of their left hand argument to the input of their right hand argument, thereby forcing the two actions (via the data dependency) to be performed in the correct order. The combinator >> throws away the result of its first argument, while >>= takes the result of its first argument and passes it on to its second argument. The similarity of monadic I/O-performing programs to imperative programs is no surprise: when performing I/O we specifically want to impose a total order on I/O operations.

It is often also useful to have an action that performs no I/O, and immediately returns a specified value:

```
return :: a -> IO a
```

For example, an echo action that reads a character, prints it, and returns the character read, might look like this:

```
echo :: IO Char
echo = readChar       >>= \c ->
       writeChar c    >>
       return c
```

As well as performing input/output, we also provide actions to create new mutable variables, and then to read and write them. The relevant primitives are [2]:

```
newMutVar   :: MutVar a
readMutVar  :: MutVar a -> IO a
writeMutVar :: MutVar a
```

[1] The notation \c->E, for some expression E, denotes a lambda abstraction. In Haskell, the scope of a lambda abstraction extends as far to the right as possible; in this example the body of the \c-abstraction includes everything after the \c.

[2] In reality the types a little more general than these, allowing state-manipulating computations to be encapsulated, but we omit these details here. They can be found in Launchbury & Peyton Jones [1994].

A value of type `MutVar t` can be thought of as the name of, or reference to, a mutable location in the state that holds a value of type `t`. This location can be modified with `writeMutVar` and read with `readMutVar`.

So far we have shown how to build larger actions out of smaller ones, but how do actions ever get performed — that is, applied to the real world? Every program defines a value `main` that has type `IO ()`. The program can then be run by applying `main` to the state of the world. For example, a complete program that reads and echos a single line of input is:

```
main :: IO ()
main = echo   >>= \c ->
       if c == '\n'
       then return ()
       else main
```

In principle, then, a program is just a state transformer that is applied to the real world to give a new world. In practice, however, it is crucial that the side-effects the program specifies are performed *incrementally*, and not all at once when the program finishes. A state-transformer semantics for I/O is therefore, alas, unsatisfactory, and becomes untenable when concurrency is introduced, a matter to which we return in Section 7.

More details of monadic I/O and state transformers can be found in Launchbury & Peyton Jones [1996], Launchbury & Peyton Jones [1994], Peyton Jones & Wadler [1993]

3.2 Processes

Concurrent Haskell provides a new primitive `forkIO`, which starts a concurrent process[3]:

```
forkIO :: IO a -> IO a
```

`forkIO a` is an action which takes an action, `a`, as its argument and spawns a concurrent process to perform that action. The I/O and other side effects performed by `a` are interleaved in an unspecified fashion with those that follow the `forkIO`. Here's an example:

```
let
  -- loop ch prints an infinite sequence of ch's
  loop ch = putChar ch >> loop ch
in
forkIO (loop 'a')     >>
loop 'z'
```

The `forkIO` spawns a process which performs the action `loop 'a'`. Meanwhile, the "parent" process continues on to perform `loop 'z'`. The result is that an infinite sequence of interleaved `'a'`s and `'z'`s appears on the screen; the exact interleaving is unspecified (but see Section 7.1).

As a more realistic example of `forkIO` in action, our mail tool might incorporate the following loop:

```
mailLoop :: IO ()
mailLoop
  = getButtonPress    >>= \ b ->
    case b of
      Compose -> forkIO doCompose >>
                 mailLoop
```

[3] We use the term *process* to distinguish explicit concurrency from implicit parallelism, for which we use the term *threads*. A process is managed by the Haskell runtime system, and certainly does not correspond to a Unix process.

```
    ...other things

doCompose :: IO ()     -- Pop up and manage Compose window
doCompose = ...
```

Here, `getButtonPress` is very like `getChar`; it awaits the next button press and then delivers a value indicating which button was pressed. This value is then scrutinised by the `case` expression. If its value is `Compose`, then the action `doCompose` is forked to handle the composition window, while the main process continues with the next `getButtonPress`.

The following features of `forkIO` are worth noting:

1. Because our implementation of Haskell uses lazy evaluation, `forkIO` immediately requires that the underlying implementation supports inter-process synchronisation. Why? Because a process might try to evaluate a thunk (or suspension) that is already being evaluated by another process, in which case the former must be blocked until the latter completes the evaluation and overwrites the thunk with its value.

2. Since the parent and child processes may both mutate (parts of) the same shared state (namely, the world), `forkIO` immediately introduces non-determinism. For example, if one process decides to read a file, and the other deletes it, the effect of running the program will be unpredictable. Whilst this non-determinism is not desirable, it is not avoidable; indeed, every concurrent language is non-deterministic. The only way to enforce determinism would be by somehow constraining the two processes to work on separate parts of the state (different files, in our example). The trouble is that *essentially all the interesting applications of concurrency involve the deliberate and controlled mutation of shared state*, such as screen real estate, the file system, or the internal data structures of the program. The right solution, therefore, is to provide mechanisms which allow (though alas they cannot enforce) the safe mutation of shared state, a matter to which we return in the next subsection.

3. `forkIO` is asymmetrical: when a process executes a `forkIO`, it spawns a child process that executes concurrently with the continued execution of the parent. It would have been possible to design a symmetrical fork, an approach taken by Jones & Hudak [1993]:

   ```
   symFork :: IO a -> IO b -> IO (a,b)
   ```

   The idea here is `symFork p1 p2` is an action that forks two processes, `p1` and `p2`. When both complete, the `symFork` pairs their results together and returns this pair as its result. We rejected this approach because it *forces* us to synchronise on the termination of the forked process. If the desired behaviour is that the forked process lives as long as it desires, then we have to provide the whole of the rest of the parent as the other argument to `symFork`, which is extremely inconvenient.

4. In common with most process calculi, but unlike Unix, the forked process has no name. We cannot, therefore, provide operators to wait for its termination or to kill it. The former is easily simulated (using an `MVar`, introduced next), while the latter introduces a host of new difficulties (what if the process is in the middle of an atomic action?).

5. As well as spawning `m`, `forkIO m` returns the result of `m` or, more operationally, it returns a pointer to a suspension which will in due course evaluate to the result of `m`. In this way, the child process can return a result to the parent. In practice we find that this is not very useful, and it gives rise to some semantic complications, so we are considering giving `forkIO` the simpler type

   ```
   forkIO :: IO () -> IO ()
   ```

3.3 Synchronisation and communication

At first we believed that `forkIO` alone would be sufficient to support concurrent programming in Haskell, provided that the underlying implementation correctly handled the synchronisation between two processes that try to evaluate the same thunk. Our belief was based on the idea that two processes could communicate *via* a lazily-evaluated list, produced by one and consumed by the other. Whilst processes can indeed communicate in this way, we found at least three distinct reasons to introduce additional mechanisms for synchronisation and communication between processes:

1. Processes may need exclusive access to real-world objects such as files. The straightforward way to implement such exclusive access requires a shared, mutable lock variable or semaphore.

2. How can a server process read a stream of values produced by more than one client process? One way to solve this is to provide a non-deterministic merge operation, but that is quite a sophisticated operation to provide as a primitive. Worse, it is far from clear that the quest ends there; for example, one might also want several server processes to service a single stream of requests, which seems to require a non-deterministic split primitive. We wanted to find some very simple truly-primitive operations that can be used to implement non-deterministic merge, and split, and anything else we might desire.

3. Writing stream-processing programs is throughly awkward, especially if a function consumes several streams and produces several others, as well as performing input/output. One of the reasons that monadic I/O has become so popular is precisely because stream-style I/O is so tiresome to program with. It would be ironic if Concurrent Haskell re-introduced stream processing for inter-process communication just as monadic I/O abolished it for input/output! We wanted to find a way to make communication between processes look just as convenient as I/O; indeed, from the point of view of any particular process the other processes might just as well be considered part of the external world.

One alternative we considered was to introduce channels as a new primitive data type, along with send and receive primitives. This would satisfy (2) and (3) directly, and allow (1) to be modelled using a channel. But (buffered) channels are relatively expensive beasts, and their expressiveness is not always required.

Our final solution is to combine our work on mutable state (Launchbury & Peyton Jones [1994]) with the I-structures and M-structures of the dataflow language Id (Arvind, Nikhil & Pingali [1989]; Barth, Nikhil & Arvind [1991]). First of all we have a new primitive type:

```
type MVar a
```

A value of type `MVar t`, for some type `t`, is the name of a mutable location that is either *empty* or *contains a value* of type `t`. We provide the following primitive operations on `MVars`:

`newMVar :: IO (MVar a)` creates a new `MVar`.

`takeMVar :: MVar a -> IO a` blocks until the location is non-empty, then reads and returns the value, leaving the location empty.

`putMVar :: MVar a -> a -> IO ()` writes a value into the specified location. If there are one or more processes blocked in `takeMVar` on that location, one is thereby allowed to proceed. It is an error to perform `putMVar` on a location which already contains a value. (Another alternative would have been to make `putMVar` block in this case, but that is more expensive to implement without making the language more expressive.)

A useful derived operation is `swapMVar`:

```
swapMVar :: MVar a -> a -> IO a
swapMVar var new = takeMVar var       >>= \ old ->
                   putMVar var new    >>
                   return old
```

The type `MVar` can be seen in three different ways:

- It can be seen as a synchronised version of the type `MutVar` introduced in Section 3.1.
- It can be seen as the type of channels, with `takeMVar` and `putMVar` playing the role of receive and send.
- A value of type `MVar ()` can be seen as a binary semaphore, with the signal and wait operations implemented by `putMVar` and `takeMVar` respectively.

4 A standard abstraction: buffering

A good way to understand a concurrency construct is by means of examples. The following sections describe how to implement a number of standard abstractions using `MVars`: using standard examples (such as buffering) allows easy comparison with the literature.

The first example is usually a memory cell, but of course an `MVar` implements that directly. Another common example is a semaphore, but an `MVar` implements that directly too.

4.1 A buffer variable

An `MVar` can very nearly be used to mediate a producer/consumer connection: the producer puts items into the `MVar` and the consumer takes them out. The fly in the ointment is, of course, that there is nothing to stop the producer over-running, and writing a second value before the consumer has removed the first.

This problem is easily solved, by using a second `MVar` to handle acknowledgements from the consumer to the producer. We call the resulting abstraction a `CVar` (short for channel variable).

```
type CVar a = (MVar a,          -- Producer -> consumer
               MVar ())         -- Consumer -> producer

newCVar :: IO (CVar a)
newCVar
  = newMVar                     >>= \ data_var ->
    newMVar                     >>= \ ack_var ->
    putMVar ack_var ()          >>
    return (data_var, ack_var)

putCVar :: CVar a -> a -> IO ()
putCVar (data_var,ack_var) val
  = takeMVar ack_var  >>
    putMVar data_var val

getCVar :: CVar a -> IO a
getCVar (data_var,ack_var)
```

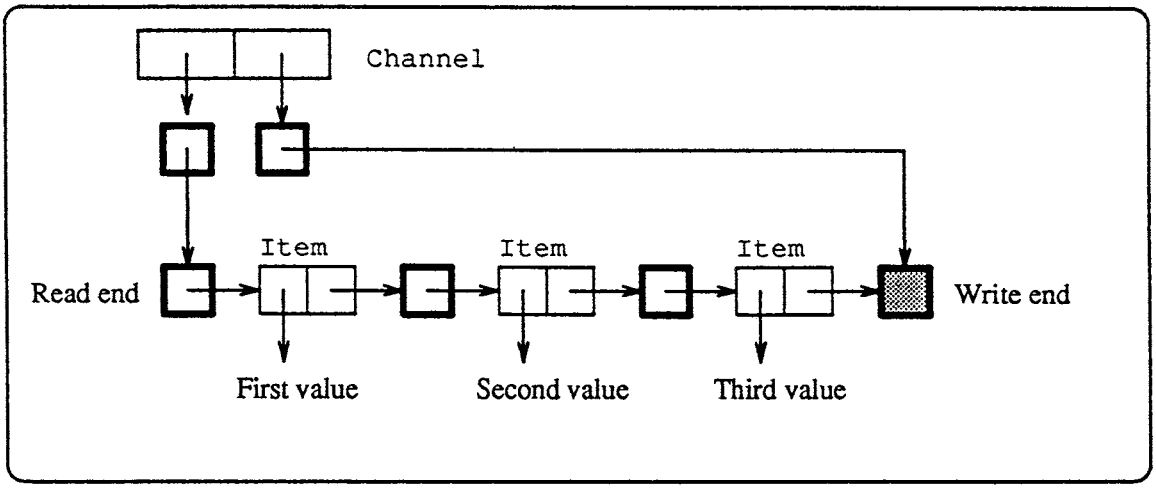

Figure 1: A channel with unbounded buffering

```
= takeMVar data_var          >>= \ val ->
  putMVar ack_var ()         >>
  return val
```

4.2 A buffered channel

A `CVar` can contain but a single value. Next, we show how to implement a channel with unbounded buffering, along with some variants. Its interface is as follows:

```
type Channel a
newChan :: IO (Channel a)
putChan :: Channel a -> a -> IO ()
getChan :: Channel a -> IO a
```

The channel should permit multiple processes to write to it, and read from it, safely.

The implementation is illustrated in Figure 1. The channel is represented by a pair of `MVars` (drawn as small boxes with thick borders), that hold the read end and write end of the buffer:

```
type Channel a = (MVar (Stream a),     -- Read end
                  MVar (Stream a))     -- Write end (the hole)
```

The `MVars` in a `Channel` are required so that channel put and get operations can atomically modify the write and read end of the channels respectively. The data in the buffer is held in a `Stream`; that is, an `MVar` which is either empty (in which case there is no data in the `Stream`), or holds an `Item`:

```
type Stream a = MVar (Item a)
```

An `Item` is just a pair of the first element of the `Stream` together with a `Stream` holding the rest of the data:

```
data Item a = Item a (Stream a)
```

A `Stream` can therefore be thought of as a list, consisting of alternating `Items` and full `MVars`, terminated with a "hole" consisting of an empty `MVar`. The write end of the channel points to this hole.

Creating a new channel is now just a matter of creating the read and write MVars, plus one (empty) MVar for the stream itself:

```
newChan = newMVar              >>= \read ->
          newMVar              >>= \write ->
          newMVar              >>= \hole ->
          putMVar read hole    >>
          putMVar write hole   >>
          return (read,write)
```

Putting into the channel entails creating a new empty Stream to become the hole, extracting the old hole and replacing it with the new hole, and then putting an Item in the old hole.

```
putChan (read,write) val
  = newMVar                                  >>= \new_hole ->
    takeMVar write                           >>= \old_hole ->
    putMVar write new_hole                   >>
    putMVar old_hole (Item val new_hole)
```

Getting an item from the channel is similar. Notice that getChan may block at the second takeMVar if the channel is empty, until some other process does a putChan.

```
getChan (read,write)
  = takeMVar read              >>= \cts ->
    takeMVar cts               >>= \(Item val new_cts) ->
    putMVar read new_cts       >>
    return val
```

It is worth noting that any number of processes can safely write into the channel and read from it. The values written will be merged in (non-deterministic, scheduling-dependent) arrival order, and each value read will go to exactly one process.

Other variants are readily programmed. For example, consider a multi-cast channel, in which there are multiple readers, each of which should see all the values written to the channel. All that is required is to add a new operation:

```
dupChan :: Channel a -> IO (Channel a)
```

The idea is that the channel returned by dupChan can be read independently of the original, and sees all (and only) the data written to the channel after the dupChan call. The implementation is simple, since it amounts to setting up a separate read pointer, initialised to the current write pointer:

```
dupChan (read,write)
  = newMVar                    >>= \ new_read ->
    takeMVar write             >>= \ hole ->
    putMVar write hole         >>
    putMVar new_read hole      >>
    return (new_read, write)
```

Another easy modification, left as an exercise for the reader, is to add an inverse to getChan:

```
unGetChan :: Channel a -> a -> IO ()
```

5 Control over scheduling

Suppose we wanted to implement a channel with *bounded* buffering; that is, one in which the writer would block if there were more than a certain number of unread elements in the buffer. A straightforward way to implement a bounded channel would be as a pair of an unbounded channel and a quantity semaphore:

```
type BChannel a = (Channel a, QSem)
```

A quantity semaphore is an abstraction with the following interface:

```
type QSem
newQSem    :: IO QSem
waitQSem   :: QSem -> IO ()
signalQSem :: QSem -> IO ()
```

A `QSem` holds an integer, initially set to zero. `waitQSem` decrements this number, blocking if it is already zero. `signalQSem` increments the number unless there are blocked processes, in which case it frees one of them.

The `QSem` in a `BChannel` records how many available slots there are in the buffer, so it is initialised with N calls to `signalQSem`, where N is the desired maximum buffer size. Then every attempt to write into the channel calls `waitQSem` to gain permission to write, and similarly every successful read calls `signalQSem`.

5.1 Implementing quantity semaphores

It is surprisingly difficult to implement quantity semaphores in terms of binary semaphores. To illustrate the difficulty, here is a typical failed attempt. Suppose we try to represent a `QSem` like this:

```
type QSem = (MVar Int, MVar ())
```

Consider a value (n, q) of type `QSem`. The first component, n, is an `MVar` recording how many units remain in the semaphore. When a `waitQSem` finds there are no units left, it takes the `MVar` q, which plays the role of a binary semaphore, and thereby blocks. A `signalQSem` that increments the count from zero will `write` to q, so as to free the blocked process, which then repeats its `wait` operation from scratch. Of course, there is a bug here, and a subtle one at that. Suppose that two processes call `waitQSem`. Each inspects n and finds it to be zero but, before either of them waits on q, two other processes completely execute `signalQSem`. Now q is full, so one of the waiting processes will be able to continue, but the other will block indefinitely. Nor can the test-and-then-block operation be done indivisibly, for then how would the lock on the quantity be released?

It is, in fact, possible to implement a quantity semaphore using a fixed handful of binary semaphores, but the implementation is very tricky indeed and not well known (Barz [1983]). However, because we can freely allocate new `MVars`, we can give a much more straightforward implementation:

```
type QSem = MVar (Int, [MVar ()])
```

A `QSem` is an `MVar` holding a pair (so that access to the whole pair is indivisible). The `Int` plays the same role as before. The second component of the pair is a list of `MVars`, *on each of which precisely one process is blocked.* It is an invariant of `QSems` that if the quantity is non-zero then the list is empty.

If a `waitQSem` finds a zero count in the `QSem`, it creates a new, private, `MVar`, adds it to the list, puts the resulting pair back in the `QSem`'s `MVar`, and then blocks on its private `MVar`:

```
waitQSem sem
  = takeMVar sem      >>= \(avail, blocked) ->
    if avail > 0 then
      putMVar (avail-1,[])    >>
    else
      newMVar >>= \block ->
      putMVar (0, block:blocked)      >>
      takeMVar block
```

The implementation of `signalQSem` is equally easy. It simply frees one blocked process if there are any, and increments the count otherwise:

```
signalQSem sem
  = takeMVar sem      >>= \(avail, blocked) ->
    case blocked of
      [] -> putMVar (avail+1, [])
      (block:blocked') -> putMVar (0, blocked')       >>
                          putMVar block ()
```

5.2 Variable-munch quantity semaphores

An obvious generalisation of quantity semaphores is for `waitQSem` and `signalQSem` to specify how much of the resource they claim or return respectively:

```
waitQSemN   :: QSem -> Int -> IO ()
signalQSemN :: QSem -> Int -> IO ()
```

Now, `(signalQSemN s n)` is equivalent to `n` successive calls to `signalQSem`, but if `waitQSemN` were to be implemented in this way, deadlock might easily result. Why? Because two processes executing a `waitQSemN` might each claim part, but not all, of the resource they require, thereby depleting it to zero and deadlocking. So `waitQSemN` must grab all its requirement at once; if not enough is available, it must block without grabbing any.

The new problem that this raises it that we may have a set of blocked processes, each with a different resource requirement. It is easy to record this information, and use it to release only the appropriate ones:

```
type QSem = MVar (Int, [(Int, MVar ())])
```

The implementation of `waitQSemN` is essentially identical to `waitQSem`. `signalQSemN` is a bit more interesting, because it may free zero or more blocked processes:

```
signalQSemN sem n
  = takeMVar sem              >>= \(avail, blocked) ->
    free (avail+n) blocked    >>= \(avail', blocked') ->
    putMVar sem (avail', blocked')

  where
    free avail [] = return (avail,[])
    free avail ((req,block):blocked)
      = if avail > req then
              putMVar block ()        >>
              free (avail-req) blocked
```

```
    else
        free avail blocked      >>= \(avail',blocked') ->
        return (avail', block:blocked')
```

The function `free` walks down the list of blocked processes, freeing any it can, and returning the depleted resource supply and remaining blocked processes.

5.3 Priority

Suppose that many processes, some important and some less important, are blocked on a single, empty `MVar`. Concurrent Haskell does not specify which of these processes will be awakened when the `MVar` is written. How can we arrange that it is the more important ones that are awakened? It would be possible to add some sort of priority mechanism to the language, but it turns out that there is no need: exactly the same trick as we used for the quantity semaphore will work here. All that is necessary is to build an abstraction that maintains a list of blocked processes (in the form of private `MVars` on which they are blocked), each paired with its priority.

5.4 Summary

This section has demonstrated that we can readily "reify" scheduling decisions, allowing them to be performed (when desired) in the language itself. The key idea is to represent a blocked process as an empty `MVar`, so that scheduling the process can be achieved by writing to the `MVar`. Much the same trick is used in the Pict language (Pierce & Turner [1995]).

6 Choice

Most process languages provide a choice construct — ALT in occam, select in Concurrent ML, + in the π-calculus — that allows a process to determine what to do next based on which of a number of communications are ready to proceed. For example, in the π-calculus the process

```
x(v).P + y(w).Q
```

will either read a value `v` from channel `x` and then behave like `P`, or read a value `w` from channel `y` and then behave like `Q`, but not both. We say that `x?v` is the *guard* for the first alternative, and similarly `y?w` guards the second.

We do not provide a choice construct in Concurrent Haskell, for several reasons:

1. Most languages that provide choice restrict it in the following way: *alternatives can only be guarded with single primitive actions.* As Reppy persuasively argues, such a restriction interacts very badly with abstraction (Reppy [1995]). For example, we might want to guard an alternative with a call to `getChan`, without knowing anything about how `getChan` is implemented.

   Of course, lifting this restriction is not straightforward. For example, it is no good synchronising on the first primitive action performed by the guard: just because the first primitive operation (doing a `take` on the read-end `MVar`) succeeds does not mean that the `getChan` succeeds! Furthermore, if the guard can be a compound action, as `getChan` certainly is, what should be done with partially completed actions from the non-chosen alternatives?

2. In our experience, the generality of choice is rarely if ever used.

3. Implementing a general choice construct can be costly, especially in a distributed setting, and especially if guards can contain both read and write operations.

4. `MVars` already provide non-determinism, as we have seen in the case of channels with multiple writers, and can be used to build application-specific choice constructs.

In short, contrary to initial impresssions, choice is expensive to implement, rarely used in its full generality, and limits abstraction.

In the rest of this section we describe how we live without choice. In common with the programming language Pict, we distinguish *singular choice* from *iterated choice*, the latter being by far the most common in practice.

6.1 Iterated choice

A very common paradigm is for a process to service several distinct sources of work. On each iteration the server chooses one of its clients, services the request, and then returns to select a new client. Such a server would be understood by the concurrent object-oriented programming community as a concurrent object.

The important thing about iterated choice is that partially-executed guards of the alternatives that "lose" — that is, are not selected — do not need to be undone, because they can simply await the next iteration of the server.

As an example, suppose that the server is dealing with network traffic arriving from two distinct sources. The functions `get1` and `get2` get a packet from the two sources respectively; `processPacket` does whatever the server does to the packet:

```
get1,get2 :: IO Packet
processPacket :: Packet -> IO ()
```

Of course, `get1` and `get2` can be as complicated as necessary. They might consist of a large series of I/O interactions, not just one primitive operation.

We can program the server by using a `CVar` as a rendezvous buffer. The server simply reads packets from this buffer. Before it does so, it forks a process for each packet source that simply reads a packet from its source and tries to write it into the buffer.

```
server :: IO ()
server
  =   -- Create empty buffer and full token
    newCVar           >>= \buf ->

      -- Create "sucking" processes
    forkIO (suck get1 buf)    >>
    forkIO (suck get2 buf)    >>

    server_loop buf

server_loop :: CVar Packet -> IO ()
server_loop buf tok
  = getCVar buf       >>= \pkt ->
    processPacket pkt >>
    server_loop buf tok

suck :: IO () -> CVar Packet -> IO ()
```

```
suck get_op buf
  = get_op            >>= \pkt ->
    putCVar buf pkt   >>
    suck get buf
```

Of course, if the clients can be "told" how to write to the server the "suck" processes are not necessary. In practice we find that doing so often breaks the abstraction that the client presents, and hence the formulation given above is required.

6.2 Singular choice

On those occasions when we want to make a "one-off" choice among competing alternatives, we put the obligation on the programmer to make the alternatives abortable. The way we choose to express this obligation is by making the alternatives have type[4]

```
type Alternative a = Commitment a -> IO ()
type Commitment  a = IO (Maybe (a -> IO ()))
data Maybe a       = Nothing
                   | Just a
```

An alternative takes an I/O action, of type `Commitment`, as an argument, which it performs exactly when it wants to commit. This `Commitment` returns either `Nothing`, indicating that some other alternative got there first and the alternative should abort, or `Just reply` where `reply` is an action that should be applied to the result of the alternative. Exactly one alternative will receive `Just reply` when it reaches its committment point; the others will all receive `Nothing`, whereupon they carry out any necessary abort actions and then die quietly.

It is now simple to define `select`:

```
select :: [Alternative a] -> IO a

select arms
  = newMVar                              >>= \ result_var ->
    newMVar                              >>= \ committed ->
    putMVar committed
            (Just (putMVar result_var)) >>
    let
      commit = swapMVar committed Nothing
      do_arm arm = forkIO (arm commit)
    in
    mapIO (do_arm committed result) arms        >>
    takeMVar result_var
```

7 Semantics

We have already hinted that regarding a program as a purely-functional state transformer gives an inadequate semantics for input/output behaviour. For example, a program that goes into an infinite loop printing `'a'` repeatedly, would just have the value $\perp$, even though its behaviour is quite different to one that goes into an infinite loop performing no input/output.

[4] The `Maybe` type is standard in Haskell, and corresponds to `option` in Standard ML. A value of type `Maybe t` is either `Nothing` or is of the form `Just v`, where `v` has type `t`. `Maybe` types are useful for encoding values which may or may not be there.

The situation worsens when concurrency is introduced, since now multiple concurrent processes are simultaneously mutating a single state. The purely-functional state-transformer semantics becomes untenable.

A way to avoid this dilemma is to provide an I/O semantics based on *labelled transition systems*, as is done by Gordon [1994]. Equivalence of programs is captured by bisimulation, and proved using coinduction. It turns out that it is fairly straightforward to extend Gordon's approach to handle Concurrent Haskell. A desirable property is that we believe the semantics can be *stratified*, so that the (large) purely-functional fragments of the program can be reasoned about as before, separately from its I/O behaviour.

7.1 Fairness

In any real system the programmer is likely to want some fairness guarantees. What, precisely, does "fairness" mean? At least, it must imply that *no runnable process will be indefinitely delayed.*

Is that enough? No, it is not. Consider a situation in which several processes are competing for access to a single `MVar`. Assuming that no process holds the `MVar` indefinitely, it should not be possible for any of the competing processes to be denied access indefinitely. One way to avoid such indefinite denial would be to specify a FIFO order for processes blocked on an `MVar`, but that is perhaps too strong. It would be sufficient to specify that *no process can be blocked indefinitely on an* `MVar` *unless another process holds that* `MVar` *indefinitely.*

8 Implementation

The implementation of Concurrent Haskell has few surprises. It is an extension of the Glasgow Haskell Compiler (GHC), a highly-optimising compiler for Haskell.

Concurrent Haskell runs as a single Unix process, performing its own scheduling internally. Each use of `forkIO` creates a new process, with its own (heap-allocated) stack. The scheduler can be told to run either pre-emptively (time-slicing among runnable processes) or non-pre-emptively (running each process until it blocks). The scheduler only switches processes at well-defined points at the beginning of basic blocks; at these points there are no half-modified heap objects, and the liveness of all registers (notably pointers) is known.

A thunk is represented by a heap-allocated object containing a code pointer and the values of the thunk's free variables. A thunk is evaluated by loading a pointer to it into a defined register and jumping to its code. When a process begins the evaluation of a thunk, it replaces the thunk's code pointer with a special "under-evaluation" code pointer. Accordingly, any other process that attempts to evaluate that thunk while it is under evaluation will automatically jump to the "under-evaluation" code, which queues the process on the thunk. When the original process completes evaluation of the thunk it overwrites the thunk with its final value, and frees any blocked processes.

An `MVar` is represented by a pointer to a mutable, heap-allocated, location. This location includes a flag to indicate whether the `MVar` is full or empty, together with either the value itself, or a queue of blocked processes.

8.1 Other primitives

One tiresome aspect is that when a process performs ordinary Unix I/O might block the whole Concurrent Haskell program, rather than just that process, which is obviously wrong. There seems to be no easy way around this. We provide a primitive that enables a solution to be built, however:

```
waitInputFD :: Int -> IO ()
```

`waitInputFD` blocks the process until the specified Unix file descriptor has input available.

The final useful primitive we have added allows a process to go to sleep for specified number of milliseconds:

```
delay :: Int -> IO ()
```

8.2 Garbage collection

An interesting question is the following: is it ever possible to garbage-collect a process? At first its seems that the answer might be quite complicated: after all, process garbage collection is a notoriously tricky business (see, for example, Hudak [1983]).

Fortunately, it turns out to be rather easy in Concurrent Haskell. The principle is as follows: *a process can be garbage-collected only if it can perform no further side effects.* Here are two immediate consequences:

1. A runnable process cannot be garbage collected, because it might perform more I/O.
2. A process blocked on an `MVar` can be garbage-collected if that `MVar` is not accessible from another non-garbage process. Why? Because the blocked process can only be released if another process puts a value into the blocking `MVar`, and that certainly can't happen if the `MVar` is unreachable from any non-garbage process.

This leads us to a very simple modification to the garbage collector:

- When tracing accessible heap objects, treat all runnable processes as roots.
- When an `MVar` is identified as reachable, identify all the processes blocked on that `MVar` as reachable too (and hence anything reachable from them).

Like any system, this one is not perfect; for example, an `MVar` might be reachable even though no further writes to it will take place. It does, however, do as well as can be reasonably expected, and it succeeds in some common cases. For example, a server with no possibility of future clients will be garbage-collected, since it is blocked on its input `MVar` and no other process now has that `MVar`.

9 Comparison with related work

We originally borrowed the idea of `MVars` directly from Id, where they are called M- structures. Id's motivation is rather different to ours: M-structures are used to allow certainly highly-parallel algorithms to be expressed that are difficult or impossible to express without them (Barth, Nikhil & Arvind [1991]). However the basic problem they solve is identical: convenient synchronisation between parallel processes. We also share with Id the expectation that programmers should rarely, if ever, encounter `MVars`. Rather, `MVars` are the "raw iron" from which more friendly abstractions can be built.

One big difference between Concurrent Haskell and Id is that in Concurrent Haskell operations on `MVars` can only be done in the I/O monad, and cannot be performed in purely-functional contexts. In Id, since everything is eventually evaluated, side effects are permitted everywhere.

It is interesting to compare `MVars` with ordinary semaphores, when each are used to provide mutual exclusion. Using semaphores (or mutex locks in ML-threads) one must remember to claim the lock before side-effecting the data it protects; that is, the mutex *implicitly* protects the data. With an `MVar`, the protected data is *explicitly inside* the `MVar`, which means that one cannot possibly forget to claim the lock before side-effecting it! Not only that, but the connection between the lock and the data it protects is more explicit: `MVar t` rather than `(t, mutex)`. Lastly, mutual exclusion using a semaphore requires at least two mutable locations: the semaphore and the data. Using an `MVar` usually collapses these two locations into one, and thereby also reduces the number of side-effecting operations. In complex situations implicit locking may still be unavoidable, but `MVars` simplify the common case.

9.1 Synchronous vs asynchronous

Viewed as a process language, `MVars` occupy an interesting and unusual niche. Generally, concurrent languages have evolved into two groups:

- Most concurrent languages provide *synchronous* communication. Examples include: CSP(Hoare [1985]), occam (Ltd [1984]), CCS (Milner [1989]), Concurrent ML (Reppy [1992]), and the π-calculus (Milner, Parrow & Walker [1992]). A communication involves a rendezvous between the sending and receiving process. There is no implict buffering on a channel, indeed no buffering at all.

- A very few concurrent languages provide asynchronous communication as primitive. The only examples known to us are both recent: the asynchronous π-calculus (Honda & Tokoro [1992]) and Pict (Pierce & Turner [1995]). Processes communicate through channels that provide infinite buffering. Sending and receiving are therefore asynchronous: the sending process can proceed without waiting for a recipient to receive the message. The reason that asynchronous languages have been unpopular is perhaps because all the early attempts to define asynchronous langauges tried to preserve message ordering in a channel, which turns out to result in a complicated semantics. In contrast, the asynchronous π-calculus and Pict define channels as holding a *multiset* of messages rather than a *sequence*; that is, messages may be received in a different order to that in which they are sent.

Concurrent Haskell occupies an intermediate position between these two extremes. An `MVar` is, in effect, a channel that holds either zero or one messages. Receiving is asynchronous if the `MVar` is full and synchronous if it is empty. Sending is asynchronous if the `MVar` is empty (and erroneous if it is full). There is no message-ordering issue because an `MVar` can contain at most one message.

Synchronous languages have three big disadvantages. The first is that it leads to a profligate use of processes. Process calculi usually start from the premises that "everything is a process" and that "processes are cheap". In practice, processes are not so cheap. Modern processors only work well when quite a bit of process state is migrated onto the CPU (into registers or the cache), and process switches inevitably involve replacing a lot of this state. For this reason we would prefer to limit concurrency to genuinely concurrent activities, rather than use it to implement everything.

In a synchronous language, *one or more extra processes are requird to implement almost every concurrency abstraction.* For example, a buffer or a semaphore requires a process to administer it, and a multicast buffer requires an extra process for each read port. In contrast, not one of the concurrency abstractions we have presented in this paper requires any processes of its own. They do all their work under the auspices of the process that calls them. Asynchronous communication, even with only a single-item buffer, seems to allow us to program passive abstractions (such as buffers) with efficient passive objects (such as channels or `MVars`).

Not only is it inefficient to use new processes for passive objects, but it is also complicated to understand and program. Compare, for example, Reppy's implementation of a multicast buffer (Reppy [1992, Section 5.2]) with that given earlier.

A second disadvantage of synchronous languages is that choice is not optional. In the absence of `MVars` (or channels or some similar feature) choice is absolutely required to implement memory locations, buffer, servers, and so on.

The third disadvantage of synchronous communication is that it becomes much harder to implement in a distributed setting, especially where choice is involved as it must inevitably be. (Why? Because if a choice contains both sends and receives, it may play hopscotch with a similar choice in another process (Reppy [1992]).) In the Occam language these difficulties are eased by allowing only receive operations as guards in a choice. Reppy argues convincingly that this is a serious limitation, so Concurrent ML bites the bullet and allows unrestricted guards. On the other hand, when discussing distributed systems he suggests that CML is inappropriate, and instead shows how to add Linda-style tuple-spaces to CML (Reppy [1995]). Whilst this may be a good approach, an asynchronous language at least allows the possibility of uniformly extending to distributed systems.

In spirit and programming style, therefore, Concurrent Haskell is much closer to the asynchronous languages. The distinction between a one-element channel and an infinitely-buffered channel is much smaller than that between synchronous and asynchronous. Nevertheless, it is somewhat more efficient to implement an `MVar` than an infinitely-buffered channel, and of course the latter can be built from the former. One criticism that is sometimes levelled at asynchronous systems is that the first symptom of a bug is often message-buffer exhaustion, due to messages being sent without being received; the symptoms are "likely to be far removed in time (and possibly space) from the source of the problem" (Reppy [1992Section 3.3.3]). Concurrent Haskell shares with synchronous systems immediate failure on such protocol failures. Indeed, because the effect of too many `putMVars` is an error message rather than deadlock, errors may be found even more directly. (This is pure speculation at present!)

9.2 Concurrent functional languages

Two of the first functional languages providing concurrency were PFL (Holmstrom [1983]) and Amber (Cardelli [1986]). Both supported concurrency with communication along synchronous, typed channels.

John Reppy's Concurrent ML is, as the name suggests, the ML counterpart — indeed predecessor — of Concurrent Haskell (Reppy [1992]; Reppy [1991]). CML is an influential synchronous concurrent language whose war-cry is "choice without loss of abstraction". It achieves this goal using a new abstract data type of `events`, (a subset of) whose signature is:

```
type 'a chan
type 'a event

val receive  : 'a chan -> 'a event
val transmit : 'a chan -> 'a -> unit event

val guard : (unit -> 'a event) -> 'a event
val wrap  : ('a event * ('a -> 'b)) -> 'b event

val choose : 'a event list -> 'a event
val sync   : 'a event -> 'a
```

`receive` and `transmit` are the primitive events, `guard` and `wrap` add pre-synchronisiation and

post-synchronisation actions respectively to an event, `choose` combines a list of events into a single event, and `sync` actually synchronises on an event. In many ways, a CML value of type `event t` is rather like a Haskell I/O action of type `IO t`. Both are first-class values that can be synchronised on (resp. performed) repeatedly.

An important difference is that CML events contain an implicit "synchronisation point" that is a single primitive action, encapsulated in pre- and post-synchronisation actions. Haskell I/O actions have no such structure. The corresponding disadvantage is that one writes different CML code to perform a protocol depending on whether the result is simply a unit-valued function that is called to perform side effects, or an event-valued function that is activated by `sync`. The latter are not as easy to write as the former, and the mere fact of the difference might be considered as a blow to abstraction.

FACILE is another extension of ML with concurrency (Giacalone, Mishra & Prasad [1989]), though one which is quite a bit more complex than either CML or Concurrent Haskell. Like CML, FACILE employs synchronous communication.

ML-threads is a concurrency package for ML developed by Cooper & Morrisett [1990]. It provides threads, together with mutex locks and condition variables to manage thread interaction. Concurrent Haskell has a similar flavour, although it seems somewhat simpler: for example, Concurrent Haskell provides only `MVars` rather than both mutexes and condition variables.

Using Gofer, (Jones & Hudak [1993]) have recently explored issues similar to Concurrent Haskell, introducing a (symmetric) fork primitive (Section 3.2) and synchronous channels into a monadic setting. This work differs from ours in that the emphasis is on expressing parallel algorithms succintly rather than writing concurrent programs that engage in messy interaction with the outside world. Evaluating two monadic sub-computations in parallel, by 'sparking' them using a symmetric fork primitive is convenient for many parallel algorithms, but this synchronous view of process is not appropriate in the concurrent case (Section 3.2). Communication between these 'sparked' processes is done on exclusive, synchronous channels, considering it an error when more than one send occurs on a channel without a matching receive. This restriction is quite severe in a concurrent setting, as resource managers such as a window system that encapsulate and provide controlled access to some shared resource, cannot be readily expressed.

It goes without saying that we share with all of these languages the benefits of higher-order functions, polymorphic typing, the ability to pass any value along a channel (including functions, channels, and as-yet-unevaluated suspensions).

9.3 Functional operating systems

The early 1980s saw a great deal of work done on functional operating systems. Typical was the work of Jones and Henderson (Henderson [1982]; Jones [1983]; Jones [1984]), and Stoye's "sorting office" (Stoye [1985]). All of this work was based on the idea of processes communicating through streams of messages, with a non-deterministic merge primitive, or in Stoye's case an external sorting office, that provided a choice construct. Programming using streams is not particularly easy, however, requiring a great deal of tagging and untagging to keep the plumbing straight.

Cupitt's made an advance over stream processing by introducing a form of monadic I/O (actually presented using continutations), with explicit process forking much like `forkIO` (Cupitt [1992]). Communication between processes was solely by sending messages to the process; that is, every process had but a single input port through which it had to multiplex all its communication.

9.4 Concurrent object-oriented languages

Much the largest group of asynchronous concurrent languages is the that of actor languages (Agha [1986]), and concurrent object-oriented languages (Agha [1990]) such as ABCL (Yonezawa [1990]). It would be interesting to undertake a systematic comparison of them with Concurrent Haskell, but we have not yet done so .

10 Conclusions and further work

We have described a very small and simple extension to Haskell that allows concurrent programs to be written. Using this substrate we are now well advanced in the construction of a graphical user interface toolkit, Haggis (Finne & Peyton Jones [1995]). Indeed this application has been the driving force for Concurrent Haskell throughout, just as eXene was used as a test case for CML. Despite the apparently primitive nature of our single synchronisation mechanism, `MVars`, we have found the language surprisingly expressive.

One obvious topic for further work is a semantics for Concurrent Haskell, as discussed in Section 7.

Another development we are actively working on is a distributed, multiprocessor implementation of Concurrent Haskell.

Concurrent Haskell is freely available by FTP. (Connect to `ftp.dcs.glasgow.ac.uk`, look in `pub/haskell/glasgow`, and grab any version of Glasgow Haskell from 0.24 or later.)

Acknowledgements

We are grateful to Benjamin Pierce, David Turner and Luca Cardelli, who all gave us very helpful feedback on earlier versions of the paper. Thanks, too, to Jim Mattson, who implemented concurrency and `MVars` in Glasgow Haskell.

References

G Agha [1986], *Actors: a model of concurrent computation in distributed systems*, MIT Press.

G Agha [Sept 1990], "Concurrent object-oriented programming," *Comm ACM* 33, 125–141.

Arvind, RS Nikhil & KK Pingali [Oct 1989], "I-structures - data structures for parallel computing," *TOPLAS* 11, 598–632.

PS Barth, RS Nikhil & Arvind [Sept 1991], "M-structures: extending a parallel, non-strict functional language with state," in *Functional Programming Languages and Computer Architecture, Boston*, Hughes, ed., LNCS 523, Springer Verlag, 538–568.

HW Barz [Feb 1983], "Implementing semaphores by binary semaphores," *SIGPLAN Notices* 18, 39–45.

L Cardelli [1986], "Amber," in *Combinators and functional programming languages*, G Cousineau, PL Curien & B Robinet, eds., LNCS 242, Springer Verlag.

EC Cooper & JG Morrisett [Dec 1990], "Adding threads to Standard ML," CMU-CS-90-186, Dept Comp Sci, Carnegie Mellon Univ.

J Cupitt [Aug 1992], "The design and implementation of an operating system in a functional language," PhD thesis, Computing Lab, University of Kent.

S Finne & SL Peyton Jones [April 1995], "Composing Haggis," Department of Computing Science, Glasgow University.

A Giacalone, P Mishra & S Prasad [1989], "Facile: A Symmetric Integration of Concurrent and Functional Programming," *International Journal of Parallel Programming* 18.

AJ Gordon [1994], in *Functional Programming and Input/Output*, Distinguished Dissertations in Computer Science, Cambridge University Press.

P Henderson [1982], "Purely functional operating systems," in *Functional programming and its applications*, Darlington, Henderson & Turner, eds., CUP.

CAR Hoare [1985], *Communicating sequential processes*, Prentice Hall.

S Holmstrom [1983], "Polymorphic type systems and concurrent computations in functional languages," PhD thesis, Department of Computer Science, Chalmers University.

K Honda & M Tokoro [1992], "On Asynchronous Communication Semantics," in *ECOOP 91 Workshop on Object-based Concurrent Computing*, Springer-Verlag LNCS.

Paul Hudak [Aug 1983], "Distributed task and memory management," in *Symposium on Principles of Distributed Computing*, NA Lynch et al, ed., ACM, 277–289.

MP Jones & P Hudak [Aug 1993], "Implicit and explicit parallel programming in Haskell," YALEU/DCS/RR-982, Yale University.

Simon B Jones [Aug 1983], "Abstract machine support for purely functional operating systems," PRG-34, Programming Research Group, Oxford.

Simon B Jones [Sept 1984], "A range of operating systems written in a purely functional style," TR 16, Dept Comp Sci, Univ of Stirling, Companion to "Abstract machine support for purely functional operating systems". .

J Launchbury & SL Peyton Jones [1996], "State in Haskell," *Lisp and Symbolic Computation* (to appear).

J Launchbury & SL Peyton Jones [June 1994], "Lazy functional state threads," in *SIGPLAN Symposium on Programming Language Design and Implementation (PLDI'94), Orlando*, ACM.

INMOS Ltd [1984], *Occam Programming Manual*, Prentice Hall.

R Milner [1989], *Communication and concurrency*, Prentice Hall.

R Milner, J Parrow & D Walker [1992], "A calculus of mobile processes (Parts I and II)," *Information and computation* 100, 1–77.

SL Peyton Jones & PL Wadler [Jan 1993], "Imperative functional programming," in *20th ACM Symposium on Principles of Programming Languages, Charleston*, ACM, 71–84.

BC Pierce & DN Turner [1995], "Concurrent Objects in a Process Calculus," in *Theory and Practice of Parallel Programming (TPPP), Sendai, Japan*, Springer Verlag LNCS.

J Reppy [June 1992], "Higher-order concurrency," PhD thesis, TR 92-1285, Cornell University.

JH Reppy [1995], "First-class synchronous operations," AT&T Bell Laboratories.

JH Reppy [June 1991], "CML: a higher-order concurrent language," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, ACM.

W Stoye [Dec 1985], "The implementation of functional languages using custom hardware," PhD thesis, TR81, Computer Lab, University of Cambridge.

A Yonezawa, ed. [1990], *ABCL: an object-oriented concurrent system: theory, language, programming, implementation, and application*, MIT Press.

Semantics of pH: A parallel dialect of Haskell*

Shail Aditya[a] MIT
Arvind[a] MIT
Lennart Augustsson[b] Chalmers University
Jan-Willem Maessen[a] MIT
Rishiyur S. Nikhil[c] DEC, CRL

Abstract

The semantics of kernel pH are defined in the form of a parallel, normalizing interpreter. A description of I-structure and M-structure operators is also given within the same framework. Semantics of barriers in pH are presented by translation into the kernel language without barriers. The framework presented is also suitable for multithreaded compilation of pH.

1 What is pH?

pH [11] is a parallel variant of the Haskell programming language [8] with extensions for loops, synchronized side-effect operations, and explicit sequentialization. This paper discusses the operational semantics of these variations and extensions. The concrete syntax of pH is presented in the preliminary pH manual [11] and will track future Haskell versions.

There is more than one approach to parallel implementations of functional languages. For example, it is possible to exploit the implicit parallelism of a Haskell program by concurrent evaluation of the arguments of strict operators. The absence of side effects ensures that this concurrent evaluation cannot change the result. Strictness analysis techniques widen the scope of this idea, by allowing parallel evaluation of any strict function argument. Further, it is possible to provide programmer annotations to indicate sub-expressions which should be evaluated in parallel, even when the compiler cannot prove that the value of all the sub-expressions will be required. Unlike similar annotations in imperative languages, the annotations can only affect termination, and not the results (if any). All these approaches take *demand-driven* evaluation as the starting point: in the absence of annotations, an expression is evaluated only if its value is required.

An alternative approach is to use a *parallel* evaluation order which reduces all redexes in parallel, not only those whose value is required. This strategy also implements non-strict semantics, provided that no redex has its evaluation delayed indefinitely. A parallel evaluation strategy is followed by the Id programming language [10], in which any redex not in the body of a conditional or lambda abstraction is reduced. pH follows the same strategy. A brief description of pH may be given as follows.

*The research described in this paper was funded in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-92-J-1310.

[a]MIT Laboratory for Computer Science, Cambridge, MA 02139. Email: {shail,arvind,earwig}@lcs.mit.edu.

[b]Department of Computer Science, Chalmers University, Göteborg, Sweden. Email: augustss@cs.chalmers.se.

[c]Cambridge Research Laboratory, Digital Equipment Corporation, One Kendall Square, Building 700, Cambridge, MA 02139. Email: nikhil@crl.dec.com.

pH = Haskell syntax and type system + Id evaluation strategy + Id side-effect operators

In practice, implementations of pH may not guarantee that every redex is reduced eventually. This will imply that some pH programs may fail to give a result and terminate when they would do so in "traditional" Haskell. However, if pH gives a proper (non-error, non-bottom) result for a particular program then so does "traditional" Haskell, and these results are the same.

1.1 Why pH?

pH is an attempt to bring together the lazy functional community (as represented by Haskell) and the dataflow community (as represented by Id and Sisal [7]). It is hoped that by sharing a common base language, it will be easier to share ideas and implementations and freely exchange programs. By doing so, programmers need only master a single syntax and type system. There are important differences between pH and Haskell, however. By choosing Haskell as a framework, it becomes easier to isolate and therefore understand the differences between the two.

1.2 Structure of pH

pH is a layered language. pH(F) is the purely-functional subset of pH. Its main addition to Haskell is for- and while-loops. For example, one could compute the sum of the integers between 1 and n as follows.

Example 1:

```
let sum = 0
 in for i <- [1..n] do
      next sum = sum + i
    finally sum
```

Loops are purely functional, and can easily be translated into tail-recursive functions. The eager semantics of pH permits tail-recursion to be implemented more efficiently than lazy semantics.

pH(I) adds I-structures [4] to pH(F). I-structures are *single-assignment* data structures that allow fine-grain producer-consumer synchronization among parallel tasks. pH(I) preserves determinacy, *i.e.*, *confluence*, although the language is no longer referentially transparent.

pH(M) adds M-structures [6] to pH(I). M-structures are *multiple-assignment* data structures that allow mutual exclusion synchronization. M-structures introduce side-effects and non-determinacy. pH(M) also provides the ability to group statements for sequential execution using *barriers* in order to avoid unwanted race conditions among side-effect operations.

Full pH is the same as pH(M).

1.3 Outline

The rest of this paper concentrates on semantic issues in which pH differs from Haskell. Section 2 describes the parallel evaluation order of pH in the context of a kernel language and its multi-threaded interpreter. In Section 3 we extend the kernel language with I-structure and M-structure operations. Section 4 defines the semantics of barriers by translating them into the kernel language using explicit termination signals from side-effect operations. Finally, Section 5 concludes with some notes about current implementations.

c	$\in$	Constant
$f, t, x, y, z \ldots$	$\in$	Identifier
$SE, X, Y, Z, \ldots$	$\in$	Simple Expression
E	$\in$	Expression
PF^n	$\in$	Primitive Fn. with n arguments
CF^n	$\in$	Data Constructor with n arguments
S	$\in$	Statement
Constant	::=	*Integer* | *Float* | *Boolean* | `Nil`
SE	::=	Identifier | Constant
PF^1	::=	`hd` | `tl` | `select_`k
PF^2	::=	$+ \mid - \mid \cdots \mid < \mid > \mid \cdots$
CF^2	::=	`Cons`
CF^n	::=	`make_`n`_tuple`
E	::=	$SE \mid PF^n(x_1, \ldots, x_n) \mid CF^n(x_1, \ldots, x_n)$
		$\mid \lambda x.\ E \mid \texttt{case}(x, E_1, \ldots, E_n)$
		$\mid \texttt{ap}(f, z) \mid \texttt{sap}(f, z) \mid$ Block
Block	::=	$\{\ S\ \texttt{in}\ x\ \}$
S	::=	$\epsilon \mid x = E \mid S_1; \ldots; S_n$
Program	::=	Block

Figure 1: The kernel pH language.

2 Parallel Evaluation Order

pH follows an *eager* evaluation strategy: all tasks execute in parallel, restricted only by the data dependencies among them. This strategy automatically exposes large amounts of parallelism both within and across procedures. This is in contrast with a *lazy* evaluation strategy followed by Haskell: only those tasks are evaluated which are required to produce the result. This strategy imposes a sequential constraint on the overall computation, although the exact ordering of tasks is decided dynamically.

In this section we describe the eager evaluation model of pH by giving a parallel interpreter for the kernel language of pH. Our interpreter is based on a parallel abstract machine, which is in the spirit of the G-machine and its later variants [9, 5, 12]. We first describe the kernel language (Section 2.1) and then our parallel abstract machine (Section 2.2) and its instruction set (Section 2.3). It is followed by a description of the interpreter (Section 2.4).

2.1 The Kernel pH Language

The abstract syntax of the kernel pH language is shown in Figure 1. The kernel language ensures that every intermediate result of a complex expression is explicitly named using an identifier. This is convenient for expressing and preserving the sharing of subexpressions within a computation [3]. In order to simplify the description of the evaluation rules in this paper, we do not allow constants to appear as arguments to primitive functions.

Aside from the usual arithmetic primitives, the kernel language provides functions for constructing and selecting elements of lists and n-ary tuples. The language also provides n-ary case

ι	$\in$	Instruction		
ℓ	$\in$	Location		
ν	$\in$	Value	::=	Constant | Location | $\langle \lambda x.\ E, \rho \rangle$
ρ	$\in$	Environment Frame	=	Identifier $\rightarrow$ Location
σ	$\in$	Store	=	Location $\rightarrow$ {$\langle$full $\nu\rangle$ | $\langle$defer $\delta s\rangle$}
ω	$\in$	Thread	=	Code $\times$ Environments
δ	$\in$	Suspension	=	Thread
δs	$\in$	Suspensions	=	*List*(Thread)
ωs	$\in$	Work Queue	=	*List*(Thread)
		Accumulator	=	Value
ιs	$\in$	Code	=	*List*(Instruction)
ρs	$\in$	Environment	=	*List*(Environment Frame)
		Processor	=	Accumulator $\times$ Code $\times$ Environment
		Machine	=	*List*(Processor) $\times$ Work Queue $\times$ Store

	Processor			Global Memory	
State	Accumulator	Code	Environment	Work Queue	Store
Initial (starting proc.)	-	ιs_0	[]	[]	{}
Initial (other proc.)	-	[]	[]	[]	{}
Final (all proc.)	-	[]	[]	[]	σ
Error (any proc.)	storerr	ιs	ρs	ωs	σ

Where ιs_0 = [eval(Program), print, schedule]

Figure 2: Dynamic entities used by the abstract machine.

expressions which select one of the branches based on the value of the dispatch identifier, nested λ-expressions which may contain free identifiers, a general function application operator ap, and a parallel block construct that controls lexical scoping and enables precise sharing of subexpression values. The order of bindings in a block is not significant. The identifier following the in keyword in a block expression denotes the result of the block.

2.2 A Parallel Abstract Machine

Our parallel abstract machine consists of a number of sequential processors connected to a global shared memory. The important features of our machine are described in Figure 2. Each memory location can hold a value or a list of suspended threads, and is tagged with its status. A full location contains a value. A newly allocated location is tagged with defer and it is initialized to be empty. A *value* is either a constant, an allocated store location, or a function closure. The global memory holds three types of entities: heap storage, activation frames and a *work queue* which is a list of *threads*, that is, (*code*, *environment*) pairs.

The processor state consists of code, environment, and an *accumulator* which can hold a value. Any idle processor can get work by dequeuing a thread from the work queue and loading its code and environment space from the thread. Initially, it is assumed that the whole machine is empty. A program E is started by executing the following code on some processor in the empty environment.

[eval(E), print, schedule]

The eval instruction interprets an expression by structural decomposition in a given environment. Environments map program identifiers to frame locations, and are structured in a stack like manner. eval of a block allocates a new *activation frame* in the global memory to store the values of the bound identifiers of the block. It also creates a new *environment frame* for these identifiers and pushes it on the processor environment. The bound identifiers of the block point to locations in the newly allocated activation frame. New environment frames (but not activation frames) are also created in case of λ-expressions and function applications. The sharing of computations is achieved entirely through the activation frame locations: all references to an identifier within the scope of its definition lead to the same frame location. The separation of environment frames and activation frames permits environments to be copied freely and therefore allows us to treat function closures in the same way as simple values.

At each step, the processor executes the instruction at the head of the current code sequence, modifying the processor state in the process. The instruction set of the processor has instructions to load and store the accumulator, perform arithmetic and logic operations on store locations, and suspend the current thread in case of missing values. This is in contrast with lazy functional languages where unevaluated locations contain *thunks* that compute its value on-demand. Typically an instruction is popped from the code sequence after execution, but some instructions may also add new instructions to the code sequence. This is the primary way to alter the control flow. There are also instructions to push and pop environment frames, and enqueue and dequeue threads from the work queue.

The work queue is maintained as a FIFO queue in order to guarantee fair scheduling. This queue must be manipulated atomically. It is easy to modify the machine so that each processor has its own work queue which is maintained in a FIFO manner. In case a processor goes idle, it can pick up a thread from the back of the work queue of any other processor. As long as we do not arbitrarily suspend the currently running thread on any processor, this strategy still implements a fair schedule while considerably reducing the contention due to the atomic manipulation of the work queue.

The overall state of our parallel machine is described by the state of all the processors and the global memory and the work queue. Figure 2 also shows the initial, the final and the error states of the processors of our machine. The machine starts by scheduling an initial sequence of instructions on some processor that evaluates the program, prints the result present in the accumulator[1], and then continues to schedule the remaining work present in the work queue. This emphasizes the non-strict nature of this evaluation model: the program may print a result before final termination. The machine halts when both the code sequence and the work queue are empty. The machine is said to be *deadlocked* when there are suspended computations in the store but there is no work to be done. If there are no suspended computations, then the machine is said to have *terminated normally*. An error may be generated during execution due to exceptional conditions such as a type mismatch, an arithmetic overflow, or out of bounds access to data structures. However, if an attempt is made to write into a memory location that is already full then the whole machine is said to have reached an error state as shown in Figure 2.

2.3 Instruction Set

Figure 3 shows the instructions used by our interpreter. We will describe the semantics of these instructions in terms of a state transition involving a 5-tuple $(\nu, \iota s, \rho s, \omega s, \sigma)$. The first three

[1]The main thread is the only place where a print instruction may appear.

eval(E)	::	Evaluate E in the current environment and leave the result in the accumulator
touch(ℓ)	::	Check if the location ℓ is full, if not suspend the current thread
rtouch(ℓ)	::	Check if the location ℓ is full, if not suspend the current thread including the rtouch instruction. (Thus, rtouch can be retried when the thread is activated again).
load(ℓ)	::	Load the accumulator with the value present in the location ℓ (the location must be full)
take(ℓ)	::	Load the accumulator with the value present in the location ℓ and mark the location empty (the location must be full)
loadi_k(ℓ)	::	Load the accumulator with the value in the location at an offset k from the location pointed to by ℓ, or suspend the thread if the location is empty
store(ℓ)	::	Store the value in the accumulator in the location ℓ and reactivate any suspensions
storerr(ℓ)	::	Generate an error if the location ℓ is full
PF^n($\ell_1, \ldots, \ell_n$)	::	Apply strict primitive operator PF^n (such as +) to the values at locations $\ell_1, \ldots, \ell_n$ (the locations must be full)
switch($\ell_x, \iota s_1, \ldots, \iota s_n$)	::	Switch to the branch indexed by the value at location ℓ_x
pushenv(ρ)	::	Push the environment frame ρ onto the environment stack
popenv	::	Pop the top-level environment frame from the environment stack
schedule	::	Schedule the next thread from the work queue
print	::	Print the accumulator contents

Figure 3: The instruction set of the parallel abstract machine.

elements of the tuple refer to the accumulator carrying a value ν, the code sequence ιs, and the environment ρs on the processor where the instruction is executed. The last two elements of the tuple are the work queue ωs, and the dynamic store σ which are shared amongst all processors.

All the instructions that modify the global store must be executed atomically. That is, while a processor reads and modifies a given location within the store, no other processor should be allowed to read or write that location. Transactions on different locations may proceed in parallel. In case where an instruction sequence ιs has to be executed atomically, we will indicate that by writing $atomic(\iota s)$.

Touch Instructions

The touch instruction tests the status of a location. If the status is full, it does nothing. Otherwise, (status is defer) the code sequence following the current instruction is suspended at that location along with the current environment. The rtouch instruction is similar except that the touch operation is retried.

touch:

$$\frac{\langle\_ \quad \mathsf{touch}(\ell) : \iota s \quad \rho s \quad \omega s \quad \sigma[\ell \mapsto \langle \mathsf{full}\ \nu\rangle]\rangle}{\langle\_ \quad \iota s \quad \rho s \quad \omega s \quad \sigma\rangle}$$

$$\frac{\langle\_ \quad \mathsf{touch}(\ell) : \iota s \quad \rho s \quad \omega s \quad \sigma[\ell \mapsto \langle \mathsf{defer}\ \delta s\rangle]\rangle}{\langle\_ \quad [\mathsf{schedule}] \quad \rho s \quad \omega s \quad \sigma[\ell \mapsto \langle \mathsf{defer}\ (\iota s, \rho s) : \delta s\rangle]\rangle}$$

rtouch:

$$\frac{\langle\_ \quad \mathsf{rtouch}(\ell) : \iota s \quad \rho s \quad \omega s \quad \sigma[\ell \mapsto \langle \mathsf{full}\ \nu\rangle]\rangle}{\langle\_ \quad \iota s \quad \rho s \quad \omega s \quad \sigma\rangle}$$

$$\frac{\langle\_ \quad \mathsf{rtouch}(\ell) : \iota s \quad \rho s \quad \omega s \quad \sigma[\ell \mapsto \langle \mathsf{defer}\ \delta s\rangle]\rangle}{\langle\_ \quad [\mathsf{schedule}] \quad \rho s \quad \omega s \quad \sigma[\ell \mapsto \langle \mathsf{defer}\ (\mathsf{rtouch}(\ell) : \iota s, \rho s) : \delta s\rangle]\rangle}$$

Note that these instructions do not start the evaluation of the expression that will fill the location being touched. In lazy evaluation this expression is always known and a thunk for it can be stored in this location. In that case, the above instructions can be easily generalized to enqueue the thunk into the work queue.

Load and Store Instructions

The load instruction loads the contents of a given location into the accumulator. The location must already be full. The take instruction is similar to load except that it leaves the location empty.

load:

$$\frac{\langle\_ \quad \mathsf{load}(\ell) : \iota s \quad \rho s \quad \omega s \quad \sigma[\ell \mapsto \langle \mathsf{full}\ \nu\rangle]\rangle}{\langle \nu \quad \iota s \quad \rho s \quad \omega s \quad \sigma\rangle}$$

take:

$$\frac{\langle\_ \quad \mathsf{take}(\ell) : \iota s \quad \rho s \quad \omega s \quad \sigma[\ell \mapsto \langle \mathsf{full}\ \nu\rangle]\rangle}{\langle \nu \quad \iota s \quad \rho s \quad \omega s \quad \sigma[\ell \mapsto \langle \mathsf{defer}\ [\,]\rangle]\rangle}$$

loadi_k is an indexed, indirect load instruction. It desugars into touching an indexed location and then loading its value into the accumulator.

loadi_k:

$$\frac{\langle\_ \quad \mathsf{loadi\_}k(\ell_x) : \iota s \quad \rho s \quad \omega s \quad \sigma[\ell_x \mapsto \langle \mathsf{full}\ \ell\rangle]\rangle}{\langle\_ \quad \mathsf{touch}(\ell + k) : \mathsf{load}(\ell + k) : \iota s \quad \rho s \quad \omega s \quad \sigma\rangle}$$

The store instruction stores the contents of the accumulator into the given location. It also reactivates any suspensions waiting in the location by enqueuing them in the work queue.

store:

$$\frac{\langle \nu \quad \mathsf{store}(\ell) : \iota s \quad \rho s \quad \omega s \quad \sigma[\ell \mapsto \langle \mathsf{defer}\ \delta s\rangle]\rangle}{\langle \nu \quad \iota s \quad \rho s \quad \omega s +\!\!+ \delta s \quad \sigma[\ell \mapsto \langle \mathsf{full}\ \nu\rangle]\rangle}$$

Storing in a location that is already full is regarded as an error. The storerr instruction tests the status of a location and produces an error if it is already full. This instruction can precede a store instruction to avoid multiple stores to the same location.

storerr:

$$\frac{\langle \_ \quad \mathsf{storerr}(\ell) : \iota s \quad \rho s \quad \omega s \quad \sigma[\ell \mapsto \langle \mathsf{full}\ \nu \rangle] \rangle}{\langle \mathtt{storerr} \quad \iota s \quad \rho s \quad \omega s \quad \sigma \rangle}$$

$$\frac{\langle \_ \quad \mathsf{storerr}(\ell) : \iota s \quad \rho s \quad \omega s \quad \sigma[\ell \mapsto \langle \mathsf{defer}\ \delta s \rangle] \rangle}{\langle \_ \quad \iota s \quad \rho s \quad \omega s \quad \sigma \rangle}$$

Arithmetic and Logic Instructions

The standard arithmetic and logic primitives are straightforward. All of them leave the result in the accumulator.

+:

$$\frac{\langle \_ \quad +(\ell_1, \ell_2) : \iota s \quad \rho s \quad \omega s \quad \sigma[\ell_1 \mapsto \langle \mathsf{full}\ \underline{m} \rangle, \ell_2 \mapsto \langle \mathsf{full}\ \underline{n} \rangle] \rangle}{\langle \underline{m+n} \quad \iota s \quad \rho s \quad \omega s \quad \sigma \rangle}$$

Switch Instruction

The switch instruction selects one of its branches based on the value present in the location being dispatched upon.

switch $(1 \leq m \leq n)$:

$$\frac{\langle \_ \quad \mathsf{switch}(\ell_x, \iota s_1, \ldots, \iota s_n) : \iota s \quad \rho s \quad \omega s \quad \sigma[\ell_x \mapsto \langle \mathsf{full}\ \underline{m} \rangle] \rangle}{\langle \_ \quad \iota s_m +\!\!+ \iota s \quad \rho s \quad \omega s \quad \sigma \rangle}$$

Environment Manipulation Instructions

The pushenv instructions pushes the given environment frame onto the local environment stack of a processor, while the popenv instruction pops one frame from the top. The contents of the accumulator are not disturbed while popping.

pushenv:

$$\frac{\langle \_ \quad \mathsf{pushenv}(\rho) : \iota s \quad \rho s \quad \omega s \quad \sigma \rangle}{\langle \_ \quad \iota s \quad \rho : \rho s \quad \omega s \quad \sigma \rangle}$$

popenv:

$$\frac{\langle \nu \quad \mathsf{popenv} : \iota s \quad \rho : \rho s \quad \omega s \quad \sigma \rangle}{\langle \nu \quad \iota s \quad \rho s \quad \omega s \quad \sigma \rangle}$$

Thread Scheduling Instruction

The execution of a schedule instruction on a processor signifies either the termination or the suspension of a thread. The processor schedules a new thread from the head of the work queue or starts *idling* if the work queue is empty. The machine halts when all processors are idling.

schedule:

$$\frac{\langle \_ \quad [\mathsf{schedule}] \quad \rho s \quad (\iota s', \rho s') : \omega s \quad \sigma \rangle}{\langle \_ \quad \iota s' \quad \rho s' \quad \omega s \quad \sigma \rangle}$$

$$\frac{\langle \_ \quad [\mathsf{schedule}] \quad \rho s \quad [\,] \quad \sigma \rangle}{\langle \_ \quad [\,] \quad \rho s \quad [\,] \quad \sigma \rangle}$$

2.4 Expression Evaluation

In this section we describe the eval instruction which is used to destructure and evaluate expressions. It leaves the value of the expression in the accumulator.

Evaluation of a constant simply loads that constant into the accumulator.

eval constant:

$$\frac{\langle \_ \quad \mathsf{eval}(c) : \iota s \quad \rho s \quad \omega s \quad \sigma \rangle}{\langle c \quad \iota s \quad \rho s \quad \omega s \quad \sigma \rangle}$$

Evaluation of an identifier first touches it and then loads its value in the accumulator.

eval identifier:

$$\frac{\langle \_ \quad \mathsf{eval}(x) : \iota s \quad \rho s \quad \omega s \quad \sigma \rangle}{\langle \_ \quad \mathsf{touch}(\rho s(x)) : \mathsf{load}(\rho s(x)) : \iota s \quad \rho s \quad \omega s \quad \sigma \rangle}$$

In case of a strict primitive function, we touch all its arguments before executing the primitive application.

eval strict PF:

$$\frac{\langle \_ \quad \mathsf{eval}(PF^n(x_1, \ldots, x_n)) : \iota s \quad \rho s \quad \omega s \quad \sigma \rangle}{\langle \_ \quad \mathsf{touch}(\rho s(x_1)) : \ldots : \mathsf{touch}(\rho s(x_n)) : \mathsf{PF}^n(\rho s(x_1), \ldots, \rho s(x_n)) : \iota s \quad \rho s \quad \omega s \quad \sigma \rangle}$$

All data selector functions such as hd, tl, and select_k are directly implemented using the indexed, indirect load instruction loadi_k.

The non-strict constructor Cons allocates a pair of empty locations in the store and immediately returns the pointer to the first location as its value. It also pushes work into the work queue to evaluate the arguments of Cons and to fill these locations eagerly. The n-ary tuple constructor make_n_tuple operates in the same fashion. Note that under lazy evaluation we would have stored thunks into these locations which would be evaluated on demand.

eval Cons:

$$\frac{\langle \_ \quad \mathsf{eval}(\mathsf{Cons}(x_1, x_2)) : \iota s \quad \rho s \quad \omega s \quad \sigma \rangle}{\langle \ell \quad \iota s \quad \rho s \quad \omega s +\!\!+ [\omega_1, \omega_2] \quad \sigma' \rangle}$$

where $\sigma' = \sigma + \{\, \ell \mapsto \langle \mathsf{defer}\ [\,] \rangle, (\ell + 1) \mapsto \langle \mathsf{defer}\ [\,] \rangle \,\}$
$\omega_1 = ([\mathsf{eval}(x_1), \mathsf{store}(\ell), \mathsf{schedule}], \rho s)$
$\omega_2 = ([\mathsf{eval}(x_2), \mathsf{store}(\ell + 1), \mathsf{schedule}], \rho s)$

λ-expressions are evaluated to a closure value. The closure records the locations for the free identifiers of the λ-body in a new environment frame.

eval λ-expression:

$$\frac{\langle \_ \quad \mathsf{eval}(\lambda x.E) : \iota s \quad \rho s \quad \omega s \quad \sigma \rangle}{\langle \langle \lambda x.\, E, \rho \rangle \quad \iota s \quad \rho s \quad \omega s \quad \sigma \rangle}$$

where $y_1, \ldots, y_m = FV(\lambda x.E)$
$\rho = \{\, y_1 \mapsto \rho s(y_1), \ldots, y_m \mapsto \rho s(y_m) \,\}$

Evaluation of a function application proceeds by first touching the function and then applying the resulting closure to the argument. The function argument is touched prior to application only if the function application is strict (sap).

eval ap:

$$\frac{\langle\_ \quad \mathsf{eval}(\mathsf{ap}(f,z)) : \iota s \quad \rho s \quad \omega s \quad \sigma\rangle}{\langle\_ \quad \mathsf{touch}(\rho s(f)) : \mathsf{ap}(\rho s(f), \rho s(z)) : \iota s \quad \rho s \quad \omega s \quad \sigma\rangle}$$

eval sap:

$$\frac{\langle\_ \quad \mathsf{eval}(\mathsf{sap}(f,z)) : \iota s \quad \rho s \quad \omega s \quad \sigma\rangle}{\langle\_ \quad \mathsf{touch}(\rho s(f)) : \mathsf{touch}(\rho s(z)) : \mathsf{ap}(\rho s(f), \rho s(z)) : \iota s \quad \rho s \quad \omega s \quad \sigma\rangle}$$

Given a closure, the ap instruction extends the closure environment frame with the given argument location and starts the evaluation of the function body after pushing the new environment frame onto the current environment. The old environment is restored after the evaluation of the body produces a result in the accumulator[2].

ap:

$$\frac{\langle\_ \quad \mathsf{ap}(\ell_f, \ell_z) : \iota s \quad \rho s \quad \omega s \quad \sigma[\ell_f \mapsto \langle \mathsf{full}\ \langle \lambda x.\ E, \rho\rangle\rangle]\rangle}{\langle\_ \quad \mathsf{pushenv}(\rho') : \mathsf{eval}(E) : \mathsf{popenv} : \iota s \quad \rho s \quad \omega s \quad \sigma\rangle}$$

where $\rho' = \rho + \{\ x \mapsto \ell_z\ \}$

The evaluation of the case expression proceeds by first touching the dispatch expression. This must yield an integer within the range of the dispatch which is used to select the appropriate branch.

eval case:

$$\frac{\langle\_ \quad \mathsf{eval}(\mathsf{case}(x, E_1, \ldots, E_n)) : \iota s \quad \rho s \quad \omega s \quad \sigma\rangle}{\langle\_ \quad \mathsf{touch}(\rho s(x)) : \mathsf{switch}(\rho s(x), [\mathsf{eval}(E_1)], \ldots, [\mathsf{eval}(E_n)]) : \iota s \quad \rho s \quad \omega s \quad \sigma\rangle}$$

The evaluation of a block expression allocates a new activation frame in the store with a fresh location for each bound identifier of the block. It also allocates new environment frame which points to the locations in the new activation frame. This environment frame is pushed onto the current environment. Following eager evaluation, auxiliary threads are added to the work queue to evaluate each of the right-hand side expressions in the extended environment. The result identifier is also evaluated under this extended environment.

eval block:

$$\frac{\langle\_ \quad \mathsf{eval}(\{\ x_1 = E_1; \ldots; x_n = E_n \text{ in } x\ \}) : \iota s \quad \rho s \quad \omega s \quad \sigma\rangle}{\langle\_ \quad \mathsf{pushenv}(\rho) : \mathsf{eval}(x) : \mathsf{popenv} : \iota s \quad \rho s \quad \omega s +\!\!+ \omega s' \quad \sigma'\rangle}$$

where $\rho = \{\ x_1 \mapsto \ell_1, \ldots, x_n \mapsto \ell_n\ \}$

$\sigma' = \sigma + \{\ \ell_1 \mapsto \langle \mathsf{defer}\ [\,]\rangle, \ldots, \ell_n \mapsto \langle \mathsf{defer}\ [\,]\rangle\ \}$

$\omega s' = [([\mathsf{eval}(E_1), \mathsf{store}(\ell_1), \mathsf{schedule}], \rho : \rho s),$
$\ldots$
$([\mathsf{eval}(E_n), \mathsf{store}(\ell_n), \mathsf{schedule}], \rho : \rho s)]$

It is also possible to initialize the newly allocated block locations with the thunks for their respective right-hand side expressions, and not enqueue threads into the work queue. These thunks will be enqueued when the locations are touched.

3 Side-Effect Operations

Functional programming languages purposely do not permit assignment or mutable storage. Instead, such features are often offered through various implementation tricks such as by using

[2]Note that restoring the environment does not imply that the function environment can be deallocated because some threads of the function body may not have terminated yet.

Constant	::=	$\cdots \mid \bullet$
PF^1	::=	$\cdots \mid$ `iAlloc` $\mid$ `mAlloc` $\mid$ `iFetch` $\mid$ `mFetch` $\mid$ `W`
PF^2	::=	$\cdots \mid$ `iStore` $\mid$ `mStore` $\mid$ `&`
S	::=	$\cdots \mid (S_1 \text{ --- } S_2)$

Figure 4: Kernel language extensions for side-effect operations.

monadic programming techniques. Alternatively, side-effects may be introduced *via* the traditional assignment operator after specifying a precise sequential ordering on all operations (*e.g.*, Scheme and ML). Id, motivated by concerns for parallelism, offers yet another alternative for incorporating side-effects. In Id, it is possible to specify only a partial order on side-effect operations and still retain an overall consistent picture of the computation.

pH supports imperative operations on I-structure [4] and M-structure [6] objects. I-structures allow the creation of a data structure to be separated from the definition of its components: attempts to use the value of a component are automatically delayed until that component is defined; attempts to redefine a component lead to an error state. M-structures, on the other hand, are fully mutable data structures whose components can be redefined repeatedly: an `mFetch` operation reads and empties a full component; an `mStore` operation (re)defines it; two successive `mStore` operations on the same component lead to an error state unless separated by an `mFetch` operation.

For some programs in pH we need a way to sequentialize M-structure operations in order to avoid race conditions implied by its parallel evaluation order. This is accomplished by the use of *control regions* and *barriers* [2]. A control region is informally defined as a set of concurrent threads that are under the same control dependence and therefore always execute together. Threads within the same control region may execute in any order or in an interleaved manner as long as the data dependencies among them are respected. Barriers provide a mechanism to detect the termination of a set of parallel activities enclosed within a control region. A barrier (---) creates two sub-regions within a given control region — one above the barrier called the *pre-region* and the other below the barrier called the *post-region*. Intuitively, no computation within the post-region is allowed to proceed until all the side-effect computations within the pre-region have terminated. This semantics is different from those proposed in [2] where a barrier waits for the termination of *all* computations rather than just those which cause side-effects.

In the rest of this section we describe the pH extensions to deal with I-structure and M-structure operations. The semantics of barriers is described in Section 4.

3.1 Kernel Language Extensions

We extend the kernel language to incorporate side-effect operations and barriers as shown in Figure 4. We add primitives to allocate, read, and write I-structures and M-structures. The fetch and store operations directly address store locations: indexed addressing is handled separately in the kernel language.

We also extend the syntax for block bindings to allow barriers—two sets of parallel block bindings may now be sequentialized by using a barrier (---) between them. Additional constants ($\bullet$), primitive operators (`W` and `&`) and strict function application (`sap`) are used to translate barriers into an ordinary set of bindings.

3.2 Evaluation Rules for Side-Effect Operations

In this section we describe additional evaluation rules for internal primitive operators involving I-structures, M-structures, and barriers. All these operators are strict on all their inputs.

I-structure Instructions

The iAlloc instruction allocates an empty I-structure array. The iFetch and iStore instructions address an I-structure location directly; all address arithmetic must be done separately. The iFetch instruction is similar to loadi_k. The iStore instruction updates an empty location with a value and reactivates any suspended continuations. Storing to an already full location is considered to be an error.

iAlloc:

$$\frac{\langle\_ \quad \mathsf{iAlloc}(\ell_x) : \iota s \quad \rho s \quad \omega s \quad \sigma[\ell_x \mapsto \langle \mathsf{full}\ \underline{n}\rangle]\rangle}{\langle \ell \quad \iota s \quad \rho s \quad \omega s \quad \sigma'\rangle}$$

where $\sigma' = \sigma + \{\ \ell \mapsto \langle \mathsf{defer}\ [\,]\rangle, \ldots, (\ell + n - 1) \mapsto \langle \mathsf{defer}\ [\,]\rangle\ \}$

iFetch:

$$\frac{\langle\_ \quad \mathsf{iFetch}(\ell_x) : \iota s \quad \rho s \quad \omega s \quad \sigma[\ell_x \mapsto \langle \mathsf{full}\ \ell\rangle]\rangle}{\langle\_ \quad \mathsf{touch}(\ell) : \mathsf{load}(\ell) : \iota s \quad \rho s \quad \omega s \quad \sigma\rangle}$$

iStore:

$$\frac{\langle\_ \quad \mathsf{iStore}(\ell_x, \ell_z) : \iota s \quad \rho s \quad \omega s \quad \sigma[\ell_x \mapsto \langle \mathsf{full}\ \ell\rangle]\rangle}{\langle\_ \quad \mathit{atomic}([\mathsf{storerr}(\ell), \mathsf{load}(\ell_z), \mathsf{store}(\ell)]) : \iota s \quad \rho s \quad \omega s \quad \sigma\rangle}$$

M-structure Instructions

The mAlloc and mStore instructions are identical to the corresponding instructions for I-structures. The mFetch instruction is similar to the iFetch except that it empties the location being accessed and has to be retried if the location is already empty.

mAlloc:

$$\frac{\langle\_ \quad \mathsf{mAlloc}(\ell_x) : \iota s \quad \rho s \quad \omega s \quad \sigma[\ell_x \mapsto \langle \mathsf{full}\ \underline{n}\rangle]\rangle}{\langle \ell \quad \iota s \quad \rho s \quad \omega s \quad \sigma'\rangle}$$

where $\sigma' = \sigma + \{\ \ell \mapsto \langle \mathsf{defer}\ [\,]\rangle, \ldots, (\ell + n - 1) \mapsto \langle \mathsf{defer}\ [\,]\rangle\ \}$

mFetch:

$$\frac{\langle\_ \quad \mathsf{mFetch}(\ell_x) : \iota s \quad \rho s \quad \omega s \quad \sigma[\ell_x \mapsto \langle \mathsf{full}\ \ell\rangle]\rangle}{\langle\_ \quad \mathit{atomic}([\mathsf{rtouch}(\ell), \mathsf{take}(\ell)]) : \iota s \quad \rho s \quad \omega s \quad \sigma\rangle}$$

mStore:

$$\frac{\langle\_ \quad \mathsf{mStore}(\ell_x, \ell_z) : \iota s \quad \rho s \quad \omega s \quad \sigma[\ell_x \mapsto \langle \mathsf{full}\ \ell\rangle]\rangle}{\langle\_ \quad \mathit{atomic}([\mathsf{storerr}(\ell), \mathsf{load}(\ell_z), \mathsf{store}(\ell)]) : \iota s \quad \rho s \quad \omega s \quad \sigma\rangle}$$

4 Semantics of M-Barriers

In this section we describe the semantics of barriers by translating them into ordinary block bindings. The idea is to systematically construct a composite termination signal from all the side-effect

EXPRESSIONS

$$
\begin{array}{lll}
\mathrm{TE} & :: & \text{Expression} \rightarrow \text{Expression} \\
\mathrm{TE}[\![c]\!] & = & c, \quad \bullet \\
\mathrm{TE}[\![x]\!] & = & x, \quad \bullet \\
\mathrm{TE}[\![\mathtt{mFetch}(x)]\!] & = & \{\, y = \mathtt{mFetch}(x); \\
 & & \quad \mathtt{in}\ y, \quad \mathtt{W}(y)\,\} \\
\mathrm{TE}[\![\mathtt{mStore}(x,z)]\!] & = & \{\, y = \mathtt{mStore}(x,z); \\
 & & \quad \mathtt{in}\ y, \quad \mathtt{W}(y)\,\} \\
\mathrm{TE}[\![PF^n(x_1,\ldots,x_n)]\!] & = & PF^n(x_1,\ldots,x_n), \quad \bullet \\
\mathrm{TE}[\![\lambda x.\, E]\!] & = & \lambda x.\, \mathrm{TE}[\![E]\!], \quad \bullet \\
\mathrm{TE}[\![\mathtt{case}\ (x, E_1, \ldots, E_n)]\!] & = & \mathtt{case}\ (x, \mathrm{TE}[\![E_1]\!], \ldots, \mathrm{TE}[\![E_n]\!]) \\
\mathrm{TE}[\![\mathtt{ap}(f,x)]\!] & = & \mathtt{ap}(f,x) \\
\mathrm{TE}[\![\{\, S\ \mathtt{in}\ x\,\}]\!] & = & \{\, S'\ \mathtt{in}\ x, \quad s\,\} \\
\text{where} \quad S', s = \mathrm{TS}[\![S]\!] & &
\end{array}
$$

STATEMENTS

$$
\begin{array}{lll}
\mathrm{TS}[\![]\!] & :: & \text{Statement} \rightarrow \mathit{List}(\text{Statement}) \times \text{Identifier} \\
\mathrm{TS}[\![\epsilon]\!] & = & (s = \bullet), \quad s \\
\mathrm{TS}[\![x = E]\!] & = & (x, s = \mathrm{TE}[\![E]\!]), \quad s \\
\mathrm{TS}[\![S_1; \ldots; S_n]\!] & = & (S_1'; \ldots; S_n'; s = s_1\ \&\ \cdots\ \&\ s_n), \quad s \\
\text{where} \quad S_i', s_i = \mathrm{TS}[\![S_i]\!] & 1 \le i \le n & \\
\mathrm{TS}[\![S_1 \text{ --- } S_2]\!] & = & (S_1'; \\
 & & \quad f = \lambda s.\{\, S_2'\ \mathtt{in}\ y_1, \ldots, y_m, s_2\,\}; \\
 & & \quad y_1, \ldots, y_m, s_2' = \mathtt{sap}(f, s_1)), \quad s_2'
\end{array}
$$

$$
\begin{array}{lll}
\text{where} \quad S_i', s_i & = & \mathrm{TS}[\![S]\!] \quad 1 \le i \le 2 \\
y_1, \ldots, y_m & = & BV(S_2)
\end{array}
$$

Figure 5: Semantics of M-barriers.

operations taking place within the pre-region of a barrier (including its child regions) which would then be used to trigger the operations present in the post-region of the barrier. The notion of signal generation and composition may be understood by looking at the translation of a simple parallel block as shown below.

$$
\begin{array}{lll}
\mathrm{TE}[\![\, \{\, x_1 = E_1; & = & \{\, x_1, s_1 \ = \ \mathrm{TE}[\![E_1]\!]; \\
\quad \cdots & & \quad \cdots \\
\quad x_n = E_n; & & \quad x_n, s_n \ = \ \mathrm{TE}[\![E_n]\!]; \\
\quad \mathtt{in}\ x\,\}\,]\!] & & \quad \mathtt{in}\ x, s_1\ \&\ \cdots\ \&\ s_n\,\}
\end{array}
$$

An expression E_i is translated as $\mathrm{TE}[\![E_i]\!]$ which dynamically returns a value (bound to x_i) along with an explicit termination signal s_i. The operator W is used to detect the termination of each side-effect operation (mFetch and mStore) within E_i. The value of s_i would become $\bullet$ as soon as all the side-effect operations in E_i have terminated. The operator & is then used to combine all such signals into a single signal for the whole block.

The primitive W operator produces a signal $\bullet$ when a given identifier becomes a value. The general evaluation rule for strict primitive functions would ensure that the identifier being tested

is touched before applying the W operator. Similarly, the primitive & operator is used to combine signals from two subexpressions into one signal after they have been touched.

W:

$$\frac{\langle\text{-}\quad \mathsf{W}(\ell_x) : \iota s \quad \rho s \quad \omega s \quad \sigma[\ell_x \mapsto \langle \mathsf{full}\ \nu\rangle]\rangle}{\langle\bullet \quad \iota s \quad \rho s \quad \omega s \quad \sigma\rangle}$$

&:

$$\frac{\langle\text{-}\quad \&(\ell_1, \ell_2) : \iota s \quad \rho s \quad \omega s \quad \sigma[\ell_1 \mapsto \langle \mathsf{full}\ \bullet\rangle, \ell_2 \mapsto \langle \mathsf{full}\ \bullet\rangle]\rangle}{\langle\bullet \quad \iota s \quad \rho s \quad \omega s \quad \sigma\rangle}$$

The complete barrier translation appears in Figure 5. Note that tuple return values and non-refutable pattern matching is used only for clarity—these operations are directly desugared into primitive operators in the obvious way. The only non-trivial base cases for signal generation are those for the `mFetch` and `mStore` primitive operators. The `mFetch` operation is considered to have terminated when the value of the location being fetched is returned. Similarly, the `mStore` operation is considered to have terminated when it returns the value being stored.

In the translation of S_1 --- S_2, note that S_2 gets protected by a λ-expression, so that it does not get evaluated until the λ-expression is applied to something. That something is the termination signal of S_1, and we use sap to ensure that the application does not commence until the termination signal is available. Finally, since the bound variables of S_2 have now gone into an inner scope (in the λ-expression), we return them all and rebind them again in the outer scope.

The barrier semantics shown here are different from the semantics presented in [2] in that here we are concerned only with the termination of the side-effect operations present within the pre-region of a barrier, while the semantics presented in [2] waited for the termination of the entire computation within the pre-region.

5 Conclusion

In this paper we have presented the various semantic issues in the design of pH at which it differs from or extends the Haskell programming language. The major difference is that pH uses a parallel eager evaluation strategy as opposed to the lazy evaluation strategy of Haskell. First, we described the kernel pH language and a normalizing interpreter for it that implements this parallel evaluation order. Next, we extended the language and the interpreter with synchronizing side-effect operations, I-structures and M-structures. Finally, we described a notion of sequentialization of side-effects using barriers. We showed a systematic translation of a kernel pH program with barriers into one without barriers using primitive termination detection operators.

The readers may note that our interpreter used an explicit instruction stream rather than directly evaluating kernel language expressions. This organization allows us to define a multithreaded compilation scheme for the kernel language within the same framework. The process of compilation can be defined as simply generating all the threads statically instead of manipulating the instruction stream dynamically. It essentially gets rid of the eval instruction. Such a compilation scheme for the kernel language is described in [1].

Currently, there is a working implementation of the pH language using the Haskell/pH HBCC front-end (written in Haskell) from Chalmers University and the Id compiler Monsoon back-end (written in Lisp) from MIT. The HBCC front-end produces a kernel pH intermediate format that is converted into dataflow graphs and fed into the Monsoon back-end. Currently, we have exercised

this compiler with purely functional Haskell programs or pH programs derived by automatically transliterating Id programs. At MIT, we are currently working on a new pH back-end for commercial uniprocessor and multiprocessor architectures that is closer in spirit to the interpretation scheme described in this paper.

References

[1] Shail Aditya. Normalizing Strategies for Multithreaded Interpretation and Compilation of Non-Strict Languages. CSG Memo 374, MIT Laboratory for Computer Science, Cambridge, MA 02139, May 1995.

[2] Shail Aditya, Arvind, and Joseph E. Stoy. Semantics of Barriers in a Non-Strict, Implicitly-Parallel Language. In *Proc. Functional Programming Languages and Computer Architecture, La Jolla, CA*, Cambridge, MA 02139, June 1995. Also available as CSG Memo 367-1, MIT Lab. for Computer Sc., Cambridge, MA 02139.

[3] Zena M. Ariola and Arvind. Properties of a First-order Functional Language with Sharing. CSG Memo 347-1, Laboratory for Computer Science, MIT, Cambridge, MA 02139, June 1994. To appear in *Theoretical Computer Science*, September 1995.

[4] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. I-Structures: Data Structures for Parallel Computing. *ACM Transactions on Programming Languages and Systems*, 11(4):598–632, 1989.

[5] Lennart Augustsson and Thomas Johnsson. Parallel Graph Reduction with the $< \nu, G >$-machine. In *Proc. Fourth Intl. Conf. on Functional Programming Languages and Computer Architecture, London*, pages 202–213. ACM Press, September 1989.

[6] Paul S. Barth, Rishiyur S. Nikhil, and Arvind. M-Structures: Extending a Parallel, Non-Strict, Functional Language with State. In *Proc. Functional Programming Languages and Computer Architecture*, pages 538–568. Springer-Verlag, 1991. LNCS 523.

[7] A. P. W. Böhm, D. C. Cann, J. T. Feo, and R. R. Oldehoeft. SISAL 2.0 Reference Manual. Technical Report UCRL-MA-109098, Lawrence Livermore National Laboratory, December 1991.

[8] Paul Hudak, Simon Peyton Jones, and Philip Wadler (editors). Report on the Programming Language Haskell: A Non-strict Purely Functional Language, Version 1.2. *ACM SIGPLAN Notices*, 27(5), May 1992.

[9] Thomas Johnsson. Efficient Compilation of Lazy Evaluation. *Proc. ACM SIGPLAN '84 Symposium on Compiler Construction, SIGPLAN Notices*, 19(6):58–69, June 1984.

[10] Rishiyur S. Nikhil. Id Language Reference Manual Version 90.1. Technical Report CSG Memo 284-2, Laboratory for Computer Science, MIT, Cambridge, MA 02139, July 15 1991.

[11] Rishiyur S. Nikhil, Arvind, James Hicks, Shail Aditya, Lennart Augustsson, Jan-Willem Maessen, and Yuli Zhou. pH Language Reference Manual, Version 1.0—preliminary. CSG Memo 369, Laboratory for Computer Science, MIT, Cambridge, MA 02139, January 1995.

[12] Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, April 1992.

Monadic I/O in Haskell 1.3

Andrew D. Gordon * and Kevin Hammond †

June 1995

Abstract

We describe the design and use of monadic I/O in Haskell 1.3, the latest revision of the lazy functional programming language Haskell. Haskell 1.3 standardises the monadic I/O mechanisms now available in many Haskell systems. The new facilities allow fairly sophisticated text-based application programs to be written portably in Haskell. The standard provides implementors with a flexible framework for extending Haskell to incorporate new language features. Apart from the use of monads, the main advances over the previous standard are: character I/O based on handles (analogous to ANSI C file pointers), an error handling mechanism, terminal interrupt handling and a POSIX interface. Apart from a tutorial description of the new facilities we include a worked example: a derived monad for combinator parsing.

1 Introduction

Haskell 1.3 improves on previous versions of Haskell [12] by adopting an I/O mechanism based on *monads* [19]. This paper explains the structure of this monadic I/O mechanism, justifies some of the design decisions, and explains how to program with the new facilities. This paper provides a more in-depth treatment of I/O than is possible in the Haskell 1.3 report [9] and library documentation [10].

Previous versions of Haskell used *synchronised streams* or *dialogues* for I/O. In practice, many Haskell programmers found it cumbersome to use these constructs directly. Awkward pattern matching against the input stream was necessary, as illustrated by the program in Table 1, which simply copies its standard input to its standard output. Instead, it is common practice to use libraries of derived combinators to program at a higher level. One such library (for *continuation-passing* I/O [14, 17]) used to be part of Haskell's standard prelude.

```
main ~( Str input : ~ (Success : _ )) =
  [ ReadChan stdin,
    AppendChan stdout input
  ]
```

Table 1: Dialogue I/O in Haskell 1.2

Recently, researchers have experimented with new I/O combinators based on monads [8, 18]. These combinators are capable of capturing all the I/O operations that could be provided using the previous stream-based approach, and provide the same type security as the continuation library. The monadic approach is significantly more flexible than the other two approaches, however, in the ease with which new I/O primitives can be introduced or existing I/O primitives combined to create new combinators. Monadic I/O has proved sufficiently attractive that several Haskell systems already support at least a basic implementation, and some support sophisticated mechanisms such as inter-language working, concurrency, or direct state-manipulation.

One of the main purposes of Haskell 1.3 is to standardise primitives for monadic I/O. The design provides a basic (but "industrial-strength" and extensible) interface to common operating systems such as Unix, DOS, VMS, or the Macintosh. The design has been influenced by the I/O operations found in imperative languages. Experimental features such as graphical interfaces or mutable variables with which the Haskell community has little experience are beyond the

*University of Cambridge Computer Laboratory, New Museums Site, Cambridge, CB2 3QG, UK.

†Department of Computing Science, University of Glasgow, 17 Lilybank Gdns., Glasgow, G12 8QQ, UK.

scope of the standard. To aid backwards compatibility, the design provides a monadic interface to the majority of operations which existed in previous versions of Haskell. Some rarely-used features, such as `Binary` files, have been removed, pending better designs.

The definition of Haskell 1.3 consists of two documents. The report proper [9] defines the Haskell language and the standard prelude. The standard libraries have a separate definition [10]. Sections 2, 3 and 4 of this paper describe the contents of the I/O libraries. Section 5 shows how to write combinator parsers on top of Haskell 1.3 I/O. Section 6 outlines previous work on functional I/O and Section 7 summarises. Appendix A summarises the types of all the I/O and operating system operations provided by Haskell 1.3 and Appendix B contains code for combinator parsing.

2 Elements of Monadic I/O

Monadic I/O depends on the builtin type constructor, `IO`. An expression of some type `IO` a denotes a *computation*, that may perform I/O and then returns a result of type a. The main program (function `main` from module `Main`, which we write `Main.main`) has type `IO ()`, that is, it is a computation which performs some I/O and returns an uninteresting result. The "trivial" type `()` has only one value, the unit value, which is also written `()`. When a Haskell program runs, there is a single top-level thread of control that executes the computation denoted by `Main.main`. Only this thread of control can execute the computations denoted by expressions of monadic type.

The monad constructor `IO` is a liberation to the purist functional programmer in that it permits expression of arbitrary imperative commands within a higher-order type-secure language. Unlike languages like Lisp or ML, in which arbitrary expressions may have side-effects, only expressions of monadic type may do so in Haskell. The rest of the language is undisturbed.

Section 2.1 introduces monadic I/O using the handful of I/O operations present in the standard prelude. The majority of operations are in libraries that need to be explicitly imported by the programmer. `LibIO`, the main library, contains basic monadic functions and file handling operations. We consider simple file processing operations from `LibIO` in Section 2.2 and explain control flow and error signalling operations on the `IO` monad in Section 2.3.

2.1 Simple programs

The simplest possible programs simply output their result to the standard output device (this will normally be the user's terminal). This is done in Haskell using the `print` function.

```
print :: Text a => a -> IO ()
```

If x `::` a and type a is in the `Text` class, then `print` x is the computation that prints `show` x, a textual representation of x, on the standard output. The `Text` class contains types such as `Int`, `Bool` and `Char`, lists and tuples formed from them, and certain user-declared algebraic types. The libraries document [9] defines the `show` function and the `Text` class. Here, for example, is a program to output the first nine natural numbers and their powers of two.

```
main :: IO ()
main =  print [(n, 2^n) | n <- [0..8]]
```

The output of the program is:

```
[(0,1), (1,2), (2,4), (3,8), (4,16),
 (5,32), (6,64), (7,128), (8,256) ]
```

The `show` function, and hence also `print`, formats its output in a standard way, as in source Haskell programs, so strings and characters are quoted (for example, `"Haskell B. Curry"`), special characters are output symbolically (that is, `'\n'` rather than a newline), lists are enclosed in square brackets, and so on. There are other, more primitive functions which can be used to output literal characters or strings without quoting when this is needed (`putChar`, `putStr`). These are described in the following sections.

Interacting with the User

Haskell 1.3 continues to support Landin-stream style interaction with standard input and output, using `interact`. (The type `String` below is a synonym for `[Char]`.)

```
interact :: (String -> String) -> IO()
```

If *f* is a stream processing function, computation `interact` *f* proceeds by evaluating *f* applied to a lazy stream representing the characters available from the standard input, and printing the characters produced to standard output. For example, the following program simply removes all non-upper-case characters from its standard input and echoes the result on its standard output.

```
main = interact (filter isUpper)
```

The functions `filter` and `isUpper` come from the Haskell prelude. They have the following types.

```
filter  :: (a -> Bool) -> [a] -> [a]
isUpper :: Char -> Bool
```

When run on the following input,

```
Now is the time for all Good Men to come
to the aid of the Party.
```

this program would output the following.

```
NGMP
```

Since `interact` only blocks on input when demand arises for the lazy input stream, it supports simple interactive programs; see Frost and Launchbury [6], for instance.

Basic File I/O

```
writeFile  :: String -> String -> IO()
appendFile :: String -> String -> IO()
readFile   :: String -> IO String
```

The `writeFile` and `appendFile` functions write or append their second argument, a string, to the file named by their first argument. To write a value of any printable type, as with `print`, use the `show` function to convert the value to a string first. For example,

```
main =
  appendFile "ascii-chars"
    (show [(x,chr (x)) | x <- [0..127]])
```

writes the following to the file `ascii-chars`:

```
[(0,'\NUL'), (1,'\SOH'), (2,'\STX'), ...
... (126,'~'), (127,'\DEL')]
```

The `readFile` function reads the file named by its argument and returns the contents of the file as a string. The file is read lazily, on demand, as with `interact`.

To illustrate `readFile`, we need to compose computations in sequence. We use the infix function `(>>=)` of type `IO a -> (a -> IO b) -> IO b`. Computation *comp*1 `>>=` $\backslash x$ `->` *comp*2 begins by running *comp*1. When it returns a result x, computation *comp*2 is run, which may depend on x. For example, the following program reads the file `infile`, turns all upper-case characters into lower-case ones, and then writes the result to the file `outfile`.

```
main =
   readFile "infile"  >>= \ input ->
     let output = map toLower input
     in
         writeFile "outfile" output
```

The notation `\ p -> e` is a Haskell lambda-expression, denoting a function whose argument is the pattern `p` and whose body is the expression `e`.

This level of programming (treating files as `Strings`) was roughly all that could be done with Haskell 1.2, and in fact programs at this simple level can be used almost without change in Haskell 1.3. To write more sophisticated programs than these in Haskell 1.3, the I/O library, `LibIO`, needs to be explicitly imported.

2.2 Character-Based I/O

To process files character-by-character, Haskell 1.3 introduces *handles*, which are analogous to ANSI C's file descriptors. Stream-based operations, working on complete files or devices, such as `writeFile` or `interact`, are in fact derived from character-based primitives. The two simplest functions are `getChar` and `putChar`.

```
getChar     ::          IO Char
putChar     :: Char -> IO ()
```

The `getChar` computation reads a character *c* from the standard input device and returns *c* as its result. The `putChar` *c* computation writes character *c* to the standard output device, and returns the unit value, `()` as its result. For example, here is a program that copies its standard input character-by-character to its

standard output (equivalently to, but somewhat more verbosely than `interact id`!)

```
import LibIO

main =
   isEOF            >>= \ iseof ->
   if iseof then
     return ()
   else
     getChar        >>= \ c ->
     putChar c      >>
     main
```

This program uses several new functions. The `return` function simply returns its argument as the result of the monadic computation. The function `(>>)` is identical to `(>>=)` except that its continuation takes no argument: the result, if any, of the first computation is simply discarded. The function `isEOF` returns returns `True` when the end-of-file is reached, and `False` otherwise.

2.3 Results and Errors

I/O operations need to indicate errors without terminating the program, and implementations need to handle these errors. Hence, as well as terminating successfully with a result (for example using `return`), I/O computations may terminate in failure, returning an *error value* of the builtin type `IOError`. For instance, input operations fail with the error value `eofIOError` to indicate end of file. Users may create new error values. The function `userError` sends a string to an error value distinct from those generated by the I/O primitives. Programmers can generate failures directly via the `fail` function, of type `IOError -> IO ()`. The parsing combinators of Section 5 illustrate `fail`.

So that error values may propagate as intended, the `(>>=)` function needs to take account of the possibility of failure. If the first computation fails with some error value *e*, then the entire computation also fails with *e*.

Here is a simple parity checker to compute the parity of an input consisting of just `T`s and `F`s.

```
module Parity where
import LibIO

perr :: IOError
perr =  userError "Parity"

parity :: Bool -> IO Bool
parity b =
  isEOF                 >>= \eof
  if eof then return b

  else getChar          >>= \c ->
       if c=='T' then
       (if b then parity True
             else parity False)

  else if c=='F' || isSpace c then
    parity b

  else fail perr
```

The computation `parity True` returns `True` if the number of `T`s is even, and `False` if the number is odd. But if any character other than `T`, `F` or white-space is in the input, the computation fails with the user-defined error value `perr`.

Catching Errors

Failures can be handled by the programmer using the `catch` function, of type

```
IO a -> (IOError -> IO a) -> IO a.
```

Computation `catch` *comp f* performs computation *comp*. If *comp* returns a result x, this is the result of the entire computation. Otherwise, if *comp* returns an error value x, the computation continues with $f\ x$.

```
import LibIO; import LibSystem
import Parity

main =
  (parity True >>= print) `catch` handler
handler err =
  if err == perr then
    print "Unexpected input character" >>
    exitWith (ExitFailure 1)
  else
    fail err
```

Haskell 1.3 assumes that the operating system understands numeric return codes. Function `exitWith` sends `ExitFailure` *exitfail* to a computation that immediately terminates the Haskell program and sends the operating system the numeric code *exitfail*. Likewise, `exitWith ExitSuccess` immediately terminates Haskell and sends the code for success, the number being dependent on the operating system.

The code also shows that error values can be passed to an outer level of the program by a call to `fail` within a handler.

There is also a derived operation `try` which can be used to expose error values in computations that fail, turning the failures into successful computations. The type of `try` is `IO a -> IO (Either IOError a)`, where `Either` is a standard type defined by the following.

```
data Either a b = Left a | Right b
```

The computation `try` *comp* runs the computation *comp*, and if it returns the result x, returns result `Right` x. Otherwise if *comp* returns an error value x it returns the result `Left` x. Hence `try` *comp* never fails with an error value. Of course it may loop if *comp* loops.

Haskell also defines a similar `Maybe` type, that we will use to indicate optional results from functions and computations.

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

For example,

```
isUserError :: IOError -> Maybe String
```

determines whether an `IOError` is a user-defined error. If so it returns `Just` *err*, where *err* is a programmer-specific string. Otherwise it returns `Nothing`.

The Error Function

Haskell 1.3 continues to support the `error` function. An expression `error` *msg* can be of arbitrary type and is treated semantically as identical to a divergent expression. If such an expression is ever evaluated, implementations should halt and print the error string *msg*. The `error` function is still useful in Haskell 1.3 for indicating program bugs, for instance. The monadic error signalling mechanism is preferable for handling errors in input. There is no way to catch an error indicated by the `error` function.

3 The LibIO Library

Having explained the basic operations on the `IO` monad, the objective of this section is to cover the I/O operations provided by the `LibIO` library. We begin in Section 3.1 by defining Haskell files and handles. Section 3.2 explains how files are opened and closed. Section 3.3 explains how to control the buffering of handle I/O and Section 3.4 explains how handles may be repositioned in a file. Operations in Sections 3.5, 3.6 and 3.7 cover querying handle properties, input and output respectively. The types of all these functions are in Appendix A.

3.1 Files and Handles

Haskell interfaces to the external world through an abstract *file system*. This file system is a collection of named *file system objects*, which may be organised in *directories* (see Section 4.1). We call any file system object that isn't a directory a *file*, even though it could actually be a terminal, a disk, a communication channel, or indeed any other object recognised by the operating system. File and directory names are strings. Files can be opened, yielding a handle which can then be used to operate on the contents of that file. Directories can be searched to determine whether they contain a file system object. Files (and normally also directories) can be added to or deleted from directories.

Handles are used by the Haskell run-time system to *manage* I/O on files. They are analogous to POSIX file descriptors. A handle is a value of type `Handle`. A handle has at least the following properties:

- whether the handle manages input or output or both;
- whether the handle is *open*, *closed* or *semi-closed* (see Section 3.2);
- whether the file is seekable (see Section 3.4);
- whether buffering on the handle is disabled, or enabled on a line or block basis (see Section 3.3);
- a buffer (whose length may be zero).

Most handles will also have a current I/O position indicating where the next input or output operation will occur.

Standard Handles

There are three standard handles which manage the standard input (`stdin`), standard output, (`stdout`), and standard error devices (`stderr`),

respectively. The first two are normally connected to the user's keyboard and screen, respectively. The third, `stderr`, is often also connected to the user's screen. A separate handle is provided because it is frequently useful to separate error output from the normal user output which appears on `stdout`. In operating systems which support this separation, one or the other is often directed into a file. If an operating system doesn't distinguish between normal user output and error output, a sensible default is for the two names to refer to the same handle. It is common for the standard error handle to be *unbuffered*, so that error output appears immediately on the user's terminal, but this is not always the case—see Section 3.3.

3.2 Opening and Closing Files

The `openFile` function is used to obtain a new handle for a file. It takes a *mode* parameter of type `IOMode`, that controls whether the handle can be used for input-only (`ReadMode`), output-only (`WriteMode` or `AppendMode`), or both input and output (`ReadWriteMode`). There are I/O operations on handles similar to those provided for standard input and output. Handle operations are distinguished by the prefix `h`, as in `hGetChar`. When a file is opened for output, it's created if it doesn't already exist. If, however, the file does exist and it is opened using `WriteMode`, it is first truncated to zero length before any characters are written to it.

For instance, the copy program given earlier can be rewritten to work on files as follows.

```
import LibIO
import LibSystem

main =
   getArgs                    >>= \ args ->
   let (inf:outf:_) = args                in
   openFile inf  ReadMode     >>= \ ih   ->
   openFile outf WriteMode    >>= \ oh   ->
   copyFile ih oh             >>
   hClose ih                  >>
   hClose oh

copyFile :: Handle -> Handle -> IO ()
copyFile ih oh =
   hIsEof ih            >>= \ eof ->
   if eof then
      return ()
   else
      hGetChar ih       >>= \ c  ->
      hPutChar oh c     >>
      copyFile ih oh
```

The `getArgs` computation (whose type is `IO [String]`) returns a list of strings which are the arguments to the program. The `hClose` function closes a previously opened handle. Once closed, no further I/O can be performed on a handle. In this particular program, the two uses of `hClose` are superfluous, since all open handles are automatically closed when the program terminates. It is generally good practice to close open handles once they are finished with. Many operating systems allow a program only a limited number of live references to file system objects.

ReadWrite Mode

`ReadWriteMode` allows programmers to make small incremental changes to text files. This can be much more efficient than reading a complete file as a stream and writing this back to a new file.

Lazy Input Streams

The `hGetContents` function is used to emulate stream I/O by reading the contents of a handle lazily on demand. For example, the `interact` function can be defined by:

```
interact f =
   hGetContents stdin      >>= \ s ->
   hPutStr stdout (f s)
```

A handle becomes semi-closed as soon as it is read lazily using a `getContents` or `hGetContents` operation. In this situation, the handle is effectively closed for all purposes except lazy reading of the contents of its file, or closing the handle explicitly. If an error occurs on a semi-closed handle it is simply discarded. This is because it is not possible to inject error values into the stream of results: `hGetContents` returns a lazy list of characters, and only computations of type `IO` *a* can fail!

Normally semi-closed handles will be closed automatically when the contents of the associated stream have been read completely. Occasionally, however, the programmer may want to force a semi-closed handle to be closed before this happens, by using `hClose` (for instance if

an error occurs when reading a handle, or if the entire contents is not needed but the file must be overwritten with a new value). In such a case the contents of the lazy input list are implementation dependent.

File Locking

A frequent problem with Haskell 1.2 was that implementations were not required to lock files when they were opened. Consequently, if a program opened a file again for writing while it was still being read, the results returned from the read could be garbled. Because of lazy evaluation and implicit buffering (also not specified by Haskell 1.2), it was possible for this to happen on some but not all program executions. This problem only occurs with languages which implement lazy stream input (à la `hGetContents`) and also have non-strict semantics.

In general it is hard for programmers to avoid opening a file when it has already been opened in an incompatible way. Almost all non-trivial programs open user-supplied filenames, and there is often no way of telling from the names whether two filenames refer to the same file. The only safe thing to do is implement file locks whenever a file is opened. This could be done by the programmer if a suitable locking operation was provided, but to be secure this would need to be done on every `openFile` operation, and might also require knowledge of the operating system.

The definition requires that identical files are locked against accidental overwriting within a single Haskell program (single-writer, multiple-reader). Two physical files are certainly identical if they have the same filename, but may be identical in other circumstances. A good implementation will use operating-system level locking (mandatory or advisory), if they are appropriate, to protect the user's data files. Even so, the definition *only* requires an implementation to take precautions to avoid obvious and persistent problems due to lazy file I/O (a language feature): it *does not* require the implementation to protect against interference by other applications or the operating system itself.

File Size

For a handle *hdl* which attached to a physical file, computation `hFileSize` *hdl* returns the size of that file as an integral number of bytes. On some operating systems it is possible that this will not be an accurate indication of the number of characters that can be read from the file.

File Extents

On systems such as the Macintosh it is much more efficient to define the maximum size of a file (or extent) when it is created, and to increase this extent by the total number of bytes written if the file is appended to, rather than increasing the file size each time a block of data is written. This may allow a file to be laid out contiguously on disk, for example, and therefore accessed more efficiently. In any case, the actual file size will be no greater than the extent.

While efficient file access is a desirable characteristic, the designers felt that dealing with this aspect of I/O led to a design which was over-complex for the normal programmer. The Haskell I/O definition therefore does not distinguish between file size (the number of bytes in the file), and file extent (the amount of disk occupied by a file).

3.3 Buffering

Explicit control of buffering is important in many applications, including ones that need to deal with raw devices (such as disks), ones which need instantaneous input from the user, or ones which are involved in communication. Examples might be interactive multimedia applications, or programs such as `telnet`. In the absence of such strict buffering semantics, it can also be difficult to reason (even informally) about the contents of a file following a series of interacting I/O operations.

Three kinds of buffering are supported: line-buffering, block-buffering or no-buffering. These modes have the following effects. For output, items are written out from the internal buffer according to the buffer mode:

- **line-buffering:** the entire buffer is written out whenever a newline is output, the buffer overflows, a flush is issued, or the handle is closed.
- **block-buffering:** the entire buffer is written out whenever it overflows, a flush is issued, or the handle is closed.

- **no-buffering:** output is written immediately, and never stored in the buffer.

The buffer is emptied as soon as it has been written out.

Similarly, input occurs according to the buffer mode for handle *hdl*.

- **line-buffering:** when the buffer for *hdl* is not empty, the next item is obtained from the buffer; otherwise, when the buffer is empty, characters up to and including the next newline character are read into the buffer. No characters are available until the newline character is available.
- **block-buffering:** when the buffer for *hdl* becomes empty, the next block of data is read into the buffer.
- **no-buffering:** the next input item is read and returned.

For most implementations, physical files will normally be block-buffered and terminals will normally be line-buffered.

The computation `hSetBuffering` *hdl mode* sets the mode of buffering for handle *hdl* on subsequent reads and writes as follows.

- If *mode* is `LineBuffering`, then line-buffering is enabled if possible.
- If *mode* is `BlockBuffering` *m*, then block-buffering is enabled if possible. The size of the buffer is *n* items if *m* is `Just` *n* and is otherwise implementation-dependent.
- If *mode* is `NoBuffering`, then buffering is disabled if possible.

If the mode is changed from `BlockBuffering` or `LineBuffering` to `NoBuffering`, then

- if *hdl* is writable, the buffer is flushed as for `hFlush`;
- if *hdl* is not writable, the contents of the buffer is discarded.

The default buffering mode when a handle is opened is implementation-dependent and may depend on the object which is attached to that handle. The three buffer modes mirror those provided by ANSI C.

Flushing Buffers

Sometimes implicit buffering is inadequate, and buffers must be flushed explicitly. The computation `hFlush` *hdl* causes any items buffered for output in handle *hdl* to be sent immediately to the operating system. While it would, in principle, be sufficient to provide `hFlush` and avoid the complexity of explicit buffer setting, this would be tedious to use for any kind of buffering other than `BlockBuffering`. It would be prone to error and require programmer cooperation by providing optional flushing after each I/O operation when writing library functions.

3.4 Re-positioning Handles

Many applications need direct access to files if they are to be implemented efficiently. Examples are text editors, or simple database applications. These applications often work on read-write handles described above. It is surprising how complicated such a common and apparently simple operation as changing the I/O position is in practice. The design given here draws heavily on the ANSI C standard.

Revisiting an I/O position

On some operating systems or devices, it is not possible to seek to arbitrary locations, but only to ones which have previously been visited. For example, if newlines in text files are represented by pairs of characters (as in DOS), then the I/O position will not be the same as the number of characters which have been read from the file up to that point and absolute seeking is not sensible. Functions `hGetPosn` and `hSetPosn` together provide this functionality, using an abstract type to represent the positioning information (which may be an `Integer` or any other suitable type). There is no way to convert a `handlePosn` into an `Integer` offset. This is not generally possible. A programmer can record the current I/O position if using `hSeek`.

Seeking to a new I/O position

Operating systems such as Unix or the Macintosh allow I/O at any position in a file. The `hSeek` operation allows three kinds of positioning: absolute positioning `AbsoluteSeek`, positioning relative to the current I/O position `RelativeSeek`, and positioning relative to the

current end-of-file SeekFromEnd. Some implementations or operating systems may only support some of these operations.

All positioning offsets are an integral number of bytes. This seems to be fairly widely supported and is quite simple. The alternatives (such as defining position by the number of items which can be read from the file) seem to give designs which are difficult both to understand and to use.

3.5 Handle Properties

There are several functions that query a handle to determine its properties: hIsOpen, hIsClosed, hIsReadable, hIsSeekable and so on. Originally we considered a single operation to return all the properties of a handle. This proved to be very unwieldy, and would also have been difficult to extend to cover other properties (since Haskell does not have named records). The operation was therefore split into many component operations, one for each property that a handle must have. Determining the current I/O position is treated as a separate operation.

While there are hIsOpen and hIsClosed operations, there is no way to test whether a handle is semi-closed. This was felt to be of marginal utility for most programmers, and is easy to define if necessary.

```
hIsSemiClosed      :: Handle -> IO Bool
hIsSemiClosed h    =
   hIsOpen h                  >>= \ ho ->
   hIsClosed                  >>= \ hc ->
   return (not (ho || hc))
```

3.6 Text Input

The function hReady determines whether input is available on a handle. It is intended for writing interactive programs or ones which manage multiple input streams. Because it is nonblocking, this can lead to serious inefficiency if it is used to poll several handles.

3.7 Text Output

Most of the text output operations which are provided have already been described earlier. The distinction between hPutStr and hPutText is worth emphasis. Function hPutText outputs any value whose type is an instance of the Text class, quoting strings and characters as necessary. Function hPutStr outputs an unformatted stream of characters, so tabs appear as literal tab characters in the output and so on. For example, the following outputs the two words Hello and World on a line, separated by a tab character,

```
import LibIO
main = putStr stdout "Hello\tWorld\n"
```

whereas the following outputs the string "Hello\tWorld\n".

```
import LibIO
main = putText stdout "Hello\tWorld\n"
```

4 The Other Libraries

4.1 LibDirectory

Operations are provided in LibDirectory to

- retrieve the current working directory (getCurrentDirectory);
- set the current directory to a new directory (setCurrentDirectory);
- list the contents of a directory (getDirectoryContents);
- delete files or directories (removeFile and removeDirectory);
- and to rename files or directories (renameFile and renameDirectory).

No status operations are provided. Haskell 1.2 statusFile/statusChan were rarely, if ever, used. Their functionality is probably better provided by operating-system specific operations, which can give more exact information.

4.2 LibSystem

The LibSystem library defines a set of functions which are used to interact directly with the Haskell program's environment. The most important of these are system, which introduces a new operating system task and waits for the result of that task, and getArgs which returns the command-line arguments to the program. It is possible that neither of these functions is available on a particular system, for example, these commands do not generally make sense

under the Macintosh operating system (though they do make some sense when applications are run under command-based shells such as MPW or AppleScript). When using `system` note that the commands which are produced are operating system dependent. It is entirely possible that these commands may not be available on someone else's system, so programs which use `system` may not be portable. Here is how to create a soft-linked alias to a file under Berkeley or similar Unixes.

```
module Link where
import LibSystem
link old new =
   system ("ln -s "++old++" "++new)
```

Exit Codes

As described earlier in Section 2.3, programs can terminate immediately and return an exit code to the operating system using `exitWith`. Its argument is of type `ExitCode`, whose only constructors are `ExitSuccess` and `ExitFailure`.

Environment Variables

Simple access to environment variables is supported through the `getEnv` computation. This functionality is generally available in most operating systems in some form or other. When available it provides a useful way of communicating infrequently-changed information to a program (which it is inconvenient to specify on the command-line for shell-based systems).

4.3 LibTime and LibCPUTime

The `LibTime` library provides operations that access time and date information (useful for timestamping or for timing purposes), including simple data arithmetic and simple text output. It codifies existing practice in the shape of the `Time` library provided by `hbc`. Unlike that library it is not Unix-specific, and it provides support for international time standards, including time-zone information. Time differences are recorded in a meaningful datatype rather than as a double-precision number.

4.4 LibUserInterrupt

User-produced interrupts are the most important class of interrupt which programmers commonly want to handle. Almost all platforms, including small systems such as Macintosh and MS/DOS, provide some ability to generate user-produced interrupts.

User interrupts can be handled in Haskell if a handler is installed using `setUserInterrupt`. Whenever a user interrupt occurs, the program is stopped. If an interrupt handler is installed, this is then executed in place of the program. If no interrupt handler is installed, the program is simply terminated with an operating system failure code. For example, the following program installs an interrupt handler `ihandler` that prints `^C` on `stdout` and then continues with some new computation.

```
main = setUserInterrupt ihandler >>
       ...
ihandler = (putStr "^C") >> ...
```

4.5 LibPOSIX

A library (`LibPOSIX`) has been defined that builds on the basic monadic I/O definition to provide a complete interface to POSIX-compliant operating systems. There is insufficient space to describe this library in detail here, but the library includes facilities to manipulate file protections, control processes, handle more kinds of interrupt than `userInterrupt` etc.

5 Combinator Parsing

In this section we illustrate monadic I/O in Haskell by writing a lexer and parser for untyped lambda-calculus. Our parser recognises strings of characters input from a handle. The characters are first grouped into *tokens* by the lexer. The parser acts on the sequence of tokens.

A Lexer

A token is either an alphanumeric identifier (beginning with a letter), a special symbol from the following list,

```
symbols = "()\\="
```

or else an illegal character. Tokens are represented by the following datatype.

```
data Token
  = ALPHA String | SYMBOL Char
  | ILLEGAL Char | EoF
  deriving (Eq, Text)
```

The EoF token indicates end of file. Here is a simpler lexer.

```
hGetToken :: Handle -> IO Token
hGetToken h =
  hIsEOF h >>= \b ->
  if b then return EoF else
  hGetChar h >>= \c ->
  if isSpace c then hGetToken h else
  if isAlpha c then hGetAlpha h [c] else
  if c `elem` symbols then
    return (SYMBOL c)
  else
    return (ILLEGAL c)

hGetAlpha :: Handle -> String -> IO Token
hGetAlpha h cs =
  hIsEOF h >>= \b -> if b then
    return (ALPHA (reverse cs)) else
  hLookAhead h >>= \c ->
  if isAlphanum c then
    hGetChar h >> hGetAlpha h (c : cs)
  else
    return (ALPHA (reverse cs))
```

If *h* is a handle, `hGetToken` *h* returns *tok*, the next token readable from handle *h*. The lexer ignores whitespace when forming tokens.

Parser Combinators

We can write predictive recursive-descent parsers [2] using combinators. In a predictive parser the lookahead token unambiguously determines the recursive function to be applied at each point.

Our type of parsers is a parameterised state-transformer monad built from the `IO` monad.

```
type Parser a =
  Handle -> Token -> IO (a, Token)
```

Given a handle *h* and a lookahead token *tok*0, a parser of type `Parser` *a* may do one of three things.

Accept a phrase with result $x :: a$. The parser consumes the tokens of the phrase by calling `hGetToken` *h* and then returns (*x*,*tok*1) where *tok*1 is the new lookahead token.

Fail with a lookahead error. The parser consumes no tokens and immediately fails with an error result of the form `UserError ('L':`*msg*`)`, a *lookahead error*.

Fail with a parse error. The parser consumes some number of tokens and then fails with an error value of the form `UserError ('P':`*msg*`)`, a *parse error*.

Failure with a lookahead error is used to select alternatives based on the lookahead token; failure with a parse error indicates an unparsable input. The difference between parse and lookahead errors is coded using the first character of the error string. It would be better to use two different constructors, but there is no way for programs to extend `IOError`.

The top of Appendix B shows operations on error values. Computation `lookaheadError` *x y* immediately fails with a lookahead error indicating that *x* was expected by *y* was found. Predicate `isLookahead` determines whether an error value is a lookahead error. Whether *e* is a parse or lookahead error, computation `mkParseError` *e* turns it into a parse error and then fails with it.

The middle of Appendix B shows the implementation of the `Parser` monad. Token matching is performed by `match`. Its second argument is a predicate of type `Token -> Maybe` *a*. Given an error string *e* and a predicate *f*, parser `match` *e f* applies the predicate to the lookahead token. If the outcome is `Just` *y*, meaning that the lookahead token is accepted, then another one is obtained using `hGetToken`, and the parser's result is *y*. Otherwise if the outcome is `Nothing`, meaning that the lookahead is rejected, the parser immediately fails with a lookahead error.

If *p* and *q* are parsers, *p* `` `alt` `` *q* is the parser that accepts all the phrases accepted by either *p* or *q*, provided that the choice is determined by the lookahead token. The parser first runs parser *p*. If *p* either accepts a phrase or fails with a parse error, then so does *p* `` `alt` `` *q*. But if *p* fails with a lookahead error—in which case the lookahead is unchanged but rejected—then *q* is run instead.

Functions `returnP` and `thenP` are the two standard monadic functions, analogous to `return` and `>>=` on the `IO` monad. Parser `returnP` *x* accepts the empty phrase and returns result *x*. If parser *p* accepts a phrase with result *x*, then *p* `` `thenP` `` *f* consumes that phrase and then acts as parser *f*(*x*). Any lookahead error from *f*(*x*) must be turned into a parse error because *p* may already have consumed tokens. If

parser p fails with a lookahead or parse error, then so does p `` `thenP` `` f.

Finally, if p is a parser and h a handle, `parse` p is the computation that runs p on the tokens obtainable using `hGetToken` h.

The primitives in Appendix B are enough to build arbitrary predictive parsers. The bottom of the appendix shows some derived parser functions. Parser `theToken` *tok* accepts the token *tok* and returns it as its result. Parser `ident` accepts any alphanumeric token, and returns its `String` representation. On any other input, both these parsers fail with a lookahead error.

Function `seqP` is an unparameterised form of `thenP`; it is analogous to `>>`. Function `><` runs two parsers in sequence, and returns their results as a pair. If p is a parser, `repeatP` p applies p repeatedly until it fails with a lookahead error; it returns the list of accepted results as its result.

A Parser

Suppose we want to parse untyped lambda-calculus programs such as the following.

```
true = \(x)\(y)x
false = \(x)\(y)y

zero = \(f)\(x)x
succ = \(n)\(f)\(x)n(f)(f(x))
```

Here is a suitable grammar.

```
decl  = {ident "=" exp} EOF
exp   = ident
      | "\" "(" ident ")" exp
      | exp "(" exp ")"
```

The conventions are that X Y means X followed by Y, X | Y means X or Y, and $\{X\}$ means a possibly empty sequence of X's. The following datatype represents lambda-terms.

```
data Exp
  = VAR String | LAM String Exp
  | APP Exp Exp
  deriving Text
```

As usual, we must remove left-recursion to make the grammar suitable for recursive descent parsing.

```
decl0  = {decl1} EoF
decl1  = ident "=" exp0
exp0   = exp1 { exp2 }
exp1   = ident | "\" "(" ident ")" exp0
exp2   = "(" exp0 ")"
```

The following recursion equations represent this transformed grammar as predictive parsers.

```
decl0 =
  repeatP decl1 `thenP` \x ->
  theToken EoF `seqP` returnP x

decl1 =
  ident `thenP` \x -> eq `seqP`
  exp0 `thenP` \t -> returnP (x,t)

exp0 =
  exp1 `thenP` \t ->
  repeatP exp2 `thenP` \ts ->
  returnP (foldl APP t ts)

exp1 =
  (ident `thenP` (returnP . VAR))
  `altP`
  (lambda `seqP` lp `seqP`
   ident `thenP` \x -> rp `seqP`
   exp0 `thenP` \t ->
    returnP (LAM x t))

exp2 =
  lp `seqP` exp0 `thenP` \t ->
  rp `seqP` returnP t
```

where

```
[lp,rp,lambda,eq] =
  map (theToken . SYMBOL) symbols
```

If our main program is

```
main :: IO ()
main = parse decl0 stdin >>= print
```

here is its output on the declarations shown at the beginning of this section.

```
[("true", LAM "x" (LAM "y" (VAR "x"))),
 ("false", LAM "x" (LAM "y" (VAR "y"))),
 ("cond", LAM "b" (LAM "t" ...)),
 ("zero", LAM "f" (LAM "x" ...)),
 ("succ", LAM "n" (LAM "f" ...))]
```

Discussion

Combinator parsers—like any other recursive descent parsers—are less efficient than bottom-up table-driven parsers. But they can be quickly and simply written, and for many purposes they

are fast enough. Previous examples represented their input as a list, and hence supported arbitrary lookahead [16]. Some in addition represented their output as a list of possible parses, to cater for ambiguous grammars [3, 6, 13]. Our parsers manage their input imperatively using `hGetToken`. They are predictive—they use only a single lookahead token. They only return a single successful parse. But this suffices for many computer languages, if not natural language. Managing arbitrary lookahead would require significant re-organisation of the program. Of course, Haskell 1.3 continues to support stream-style parsing via the `interact` function. The standard prelude includes simple parsers of type

```
type ReadS a = String -> [(a,String)]
```

and pretty-printers for types in the `Text` class. The monad developed in this section shows that the combinator parsing idiom applies to imperative parsing too. Our monad is more flexible than the `ReadS` style because it allows parsing to be freely mixed with other imperative computations.

Exercises

(1) Extend the program with an evaluator for lambda-calculus terms. Use de Bruijn's name-free representation of lambda-terms, instead of the naive datatype used here. Chapter 9 of Paulson's book [16] is a good starting point.

(2) Extend the lexer to recognise numerals. Extend the grammar and parser with syntax for numerals and binary arithmetic operators.

(3) Rewrite the lexer using a Haskell array to dispatch on whether the next character is whitespace, alphabetic, symbolic or illegal.

(4) Find a grammar that can be parsed with arbitrary lookahead but not by a predictive parser.

(5) Modify the `Parser` monad to admit arbitrary lookahead. Hint: use the following definition of `Parser`, which explicitly represents lookahead errors rather than using the builtin error-handling mechanism.

```
type Parser a =
  Handle -> [Token] ->
  IO ([Token], Maybe (a, [Token]))
```

If such a parser is run on a handle h with lookahead *toks*, it returns pair (*toks1*, m) where *toks1* is the new lookahead, and m is either `Nothing` if the parse has failed or `Just` (*x*,*toks2*) if the parse was successful. In the latter case, x is the result of the parse and *toks2* is the list of tokens accepted.

6 History and Related Work

Monadic I/O dates from 1989. Cupitt [5] built a functional operational system (KAOS) in Miranda. He was the first to make large-scale use of types, similar to `IO` a, for computations returning an answer of type a. Independently, about the same time, Gordon [7] proposed a concurrent language called PFL+ with a similar type constructor. 1989 was also the year Moggi first published his theory of modular denotational semantics [15] based on the categorical notion of a strong monad. Inspired by Moggi, Wadler popularised monads as a functional programming technique for dealing with state [19]. With Peyton Jones he proposed an `IO` monad, similar to the one of this paper, for expressing I/O in Haskell [18]. Gordon's book [8] surveys previous work on functional I/O in general and monadic I/O in particular. The contribution of Haskell 1.3 is a detailed standard for portable monadic I/O in Haskell, using handles to access the file system.

There is a good deal of current work on graphical interfaces to functional languages, such as the work on Concurrent CLEAN [1] and Fudgets [4]. Graphics is beyond the intended scope of Haskell 1.3, but we would welcome any proposals for a standard monadic library for expressing graphical interfaces.

Returning an error value from a computation is analogous to raising an exception in a language like ML, except that in Haskell only expressions of `IO` type may return an error value. See Hammond's book [11] for a discussion of error values in functional languages.

6.1 Computations and Effects

The type `IO a` denotes computations in the same sense as `Integer` denotes integers and `Bool` denotes truth-values. To a first approximation at least we can think of computations as functions which take the state of the world as their argument and return a pair of an updated world and a result [18]. The main thread, defined by `Main.main`, is a sequence of state-transforming computations of type `IO a`, which directly express effects on the environment, such as character I/O, or reading and writing files. Each of the sequence of computations is applied to an implicit program state, to produce a new state together with an intermediate result. The new state and result is passed to the next computation in the sequence, and so on until the program terminates.

Within the Haskell program, expressions of type `IO a` behave identically to other expressions: they may appear evaluated or unevaluated in lists, be freely copied, and so on. Haskell expressions do not have side-effects unless they are evaluated by the top-level thread of control.

6.2 Parallelism

The interaction with parallelism is important, especially for extensions of Haskell such as the pH language. Handled carelessly, I/O could unnecessarily serialise computations and thus reduce performance. Some thought has gone into this. The semantics of I/O is serialisable in the sense that I/O operations interact with the operating system in the order they are presented at the top-level. If, however, two I/O operations do not conflict (for example, reading two different files), then it is entirely possible for them to proceed in parallel. It is still necessary, however, to ensure that error values are propagated as defined by the serial semantics. This may require a mechanism similar to that needed for controlling other speculative computations.

7 Summary

We have presented a design for I/O which has been adopted in the Haskell standard, describing some interesting aspects of the design and providing a tutorial on how it can be used effectively. Being based on the use of monads, the design is both flexible and extensible. Although only a fairly conservative basic design has been provided initially, we expect this to form the basis for more radical research departures. It already provides much useful functionality that was not previously available in Haskell 1.2.

No formal semantics for these I/O primitives is possible at present, because there is no complete formal semantics for Haskell itself. We hope in future that such a semantics will be developed. One task of such a semantics would be to show that the `IO` type does indeed form a monad in the categorical sense.

Haskell 1.3 allows programmers to write programs that can change the external or global states in an imperative fashion, but only via expressions of some type `IO a`, and only when they are then interpreted by the top-level thread of control. This contrasts with languages like LISP or ML, where expressions of any type can have side-effects. Our hope is that I/O in Haskell 1.3 will be no less expressive than in LISP or ML, and that its type system can be exploited by programmers and compilers to yield clear and efficient programs.

Acknowledgements

We are grateful to the other members of Haskell committee who have made many constructive comments on the I/O design during its period of incubation. We would also like to thank those people who have either contributed directly to the I/O design, or whose comments have had a significant impact on the design. These have included Andy Gill and Ian Poole (who both worked on previous versions of the Haskell design), Jim Mattson (who designed `LibPOSIX`), Jon Fairbairn, Ian Holyer, Kent Karlsson, Sandra Loosemoore, and Alastair Reid. We are also grateful to Will Partain and Hans Loidl for commenting on draft versions of this paper and to the anonymous referees who reviewed this paper.

References

[1] Peter Achten and Rinus Plasmeijer. The ins and outs of Concurrent Clean I/O. *Journal of Functional Programming*, 5(1), 1995.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1986.

[3] W. H. Burge. *Recursive Programming Techniques.* Addison Wesley, 1975.

[4] Magnus Carlsson and Thomas Hallgren. FUDGETS: A graphical user interface in a lazy functional language. In *FPCA'93: Conference on Functional Programming Languages and Computer Architecture, Copenhagen*, pages 321–330. ACM Press, 1993.

[5] J. Cupitt. A brief walk through KAOS. Technical Report 58, Computing Laboratory, University of Kent at Canterbury, February 1989.

[6] R. Frost and J. Launchbury. Constructing natural language interpreters in a lazy functional language. *The Computer Journal*, 32(2):108–121, April 1989.

[7] Andrew Gordon. PFL+ : A kernel scheme for functional I/O. Technical Report 160, University of Cambridge Computer Laboratory, February 1989.

[8] Andrew D. Gordon. *Functional Programming and Input/Output.* Cambridge University Press, 1994.

[9] K. Hammond et al. Report on the programming language Haskell: Version 1.3. Yale University Technical Report, to appear, June 1995.

[10] K. Hammond, J. W. Peterson, et al. Standard libraries for the programming language Haskell: Version 1.3. Yale University Technical Report, to appear, June 1995.

[11] Kevin Hammond. *PSML: a Functional Language and its Implementation in Dactl.* Pitman Press, 1991.

[12] Paul Hudak, Simon L. Peyton Jones, Philip Wadler, et al. Report on the functional programming language Haskell: A non-strict, purely functional language: Version 1.2. *ACM SIGPLAN Notices*, 27(5), March 1992. Section R.

[13] Graham Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, 2(3):323–343, July 1992.

[14] Kent Karlsson. Nebula: A functional operating system. Programming Methodology Group, Chalmers University of Technology and University of Gothenburg, 1981.

[15] Eugenio Moggi. Computational lambda calculus and monads. In *Proceedings of the 4th IEEE Symposium on Logic in Computer Science*, June 1989.

[16] Lawrence C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1991.

[17] Nigel Perry. *The Implementation of Practical Functional Programming Languages.* PhD thesis, Department of Computing, Imperial College, London, June 1991.

[18] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *Proceedings 20th ACM Symposium on Principles of Programming Languages, Charleston, South Carolina, January 1993*, pages 71–84. ACM Press, 1993.

[19] Philip Wadler. The essence of functional programming. In *Proceedings of the Nineteenth ACM Symposium on Principles of Programming Languages*, 1992.

A Summary of I/O Operations

This is an unstructured list of the fixities, types, instances, and values supported by the Haskell 1.3 I/O libraries.

```
infixr 1 >>, >>=                                        -- Prelude

type IO a                                               -- Prelude

type Handle                                             -- LibIO
type FilePath = String                                  -- LibIO

data IOMode      = ReadMode                             -- LibIO
                 | WriteMode
                 | AppendMode
                 | ReadWriteMode

data BufferMode = NoBuffering                           -- LibIO
                 | LineBuffering
                 | BlockBuffering (Maybe Int)

data HandlePosn                                         -- LibIO
data SeekMode    = AbsoluteSeek                         -- LibIO
                 | RelativeSeek
                 | SeekFromEnd

data ExitCode    = ExitSuccess                          -- LibSystem
                 | ExitFailure Int

data ClockTime                                          -- LibTime
instance Ord ClockTime                                  -- LibTime
instance Eq  ClockTime                                  -- LibTime
instance Text ClockTime                                 -- LibTime
data CalendarTime =                                     -- LibTime
     CalendarTime  Int  Int  Int  Int
                   Int  Int  Integer
                   Int  Int  String
                   Int Bool

data TimeDiff     =                                     -- LibTime
     TimeDiff Int  Int
              Int  Int  Int  Int  Integer
     deriving (Eq,Ord)

stdin, stdout, stderr :: Handle                         -- LibIO
```

Operations

The set of I/O operations is sorted alphabetically.

```
(>>=)               :: IO a     -> (a -> IO b)              -> IO b
(>>)                :: IO a     -> IO b                     -> IO b
accumulate          :: [IO a]                               -> IO [a]
addToClockTime      :: TimeDiff  -> ClockTime               -> ClockTime
appendFile          :: FilePath -> String                   -> IO ()
createDirectory     :: FilePath                             -> IO ()
diffClockTimes      :: ClockTime -> ClockTime               -> TimeDiff
either              :: (a -> c) -> (b -> c) -> (Either a b) -> c
```

```
eofIOError           ::                                          IOError
exitWith             :: ExitCode                              -> IO a
fail                 :: IOError                               -> IO a
getArgs              ::                                          IO [String]
getChar              ::                                          IO Char
getClockTime         ::                                          IO ClockTime
getCPUTime           ::                                          IO Integer
getCurrentDirectory  ::                                          IO FilePath
getDirectoryContents :: FilePath                              -> IO [FilePath]
getEnv               :: String                                -> IO String
getProgName          ::                                          IO String
handle               :: IO a     -> (IOError -> IO a)         -> IO a
hClose               :: Handle                                -> IO ()
hFileSize            :: Handle                                -> IO Integer
hFlush               :: Handle                                -> IO ()
hGetBuffering        :: Handle                                -> IO (Maybe BufferMode)
hGetChar             :: Handle                                -> IO Char
hGetContents         :: Handle                                -> IO String
hGetPosn             :: Handle                                -> IO HandlePosn
hIsClosed            :: Handle                                -> IO Bool
hIsEOF               :: Handle                                -> IO Bool
hIsOpen              :: Handle                                -> IO Bool
hIsReadable          :: Handle                                -> IO Bool
hIsSeekable          :: Handle                                -> IO Bool
hIsWritable          :: Handle                                -> IO Bool
hLookAhead           :: Handle                                -> IO Char
hPutChar             :: Handle   -> Char                      -> IO ()
hPutStr              :: Handle   -> String                    -> IO ()
hPutText             :: Text a   => Handle     -> a           -> IO ()
hReady               :: Handle                                -> IO Bool
hSeek                :: Handle   -> SeekMode   -> Integer     -> IO ()
hSetBuffering        :: Handle   -> BufferMode                -> IO ()
hSetPosn             :: HandlePosn                            -> IO ()
interact             :: (String -> String)                    -> IO ()
ioeGetHandle         :: IOError                               -> Maybe Handle
ioeGetFileName       :: IOError                               -> Maybe FilePath
isAlreadyExistsError :: IOError                               -> Bool
isAlreadyInUseError  :: IOError                               -> Bool
isFullError          :: IOError                               -> Bool
isEOFError           :: IOError                               -> Bool
isIllegalOperation   :: IOError                               -> Bool
isPermissionError    :: IOError                               -> Bool
isUserError          :: IOError                               -> Maybe String
isEOF                ::                                          IO Bool
openFile             :: FilePath -> IOMode                    -> IO Handle
print                :: Text a =>   a                         -> IO ()
putChar              :: Char                                  -> IO ()
putStr               :: String                                -> IO ()
putText              :: Text a   => a                         -> IO ()
readFile             :: FilePath                              -> IO String
removeDirectory      :: FilePath                              -> IO ()
removeFile           :: FilePath                              -> IO ()
renameDirectory      :: FilePath -> FilePath                  -> IO ()
renameFile           :: FilePath -> FilePath                  -> IO ()
return               :: a                                     -> IO a
sequence             :: [IO a]                                -> IO ()
setCurrentDirectory  :: FilePath                              -> IO ()
```

```
setUserInterrupt    :: Maybe (IO ())                -> IO (Maybe (IO ()))
system              :: String                       -> IO ExitCode
toCalendarTime      :: ClockTime                    -> CalendarTime
toUTCTime           :: ClockTime                    -> CalendarTime
toClockTime         :: CalendarTime                 -> ClockTime
try                 :: IO a                         -> IO (Either IOError a)
userError           :: String                       -> IOError
writeFile           :: FilePath -> String           -> IO ()
```

B Example: Parsing Routines

```
-- ========================================================================
-- Operations on Errors
-- ========================================================================

lookaheadError :: String -> String -> IO a
isLookahead :: IOError -> Bool
mkParseError :: IOError -> IO a

lookaheadError exp fnd =
  fail (userError "L: Expected "++exp++" but found "++fnd)
isLookahead e = case (isUserError e) of
  Just ('L':_) -> True
  _ -> False
mkParseError e = case (isUserError e) of
  Just ('L':msg) -> fail (userError ('P':msg))
  _ -> fail e

-- ========================================================================
-- Implementation of the Parser Monad
-- ========================================================================

match   :: String -> (Token -> Maybe a) -> Parser a
altP    :: Parser a -> Parser a -> Parser a
returnP :: a -> Parser a
thenP   :: Parser a -> (a -> Parser b) -> Parser b
parse   :: Parser a -> Handle -> IO a

match e f h tok0 =
  case f tok0 of
   Just x -> hGetToken h >>= \tok1 -> return (x, tok1)
   Nothing -> lookaheadError e (show tok0)
(p1 `altP` p2) h s =
  p1 h s `handle` \e ->
  if is_lookahead e then p2 h s else mkParseError e
returnP a h s = return (a, s)
(p `thenP` f) h s = p h s >>= \(a,s) -> f a h s `handle` mkParseError
parse p h = (hGetToken h >>= p h) >>= (return . fst)

-- ========================================================================
-- Derived Parser Functions
-- ========================================================================

theToken:: Token -> Parser Token
ident   :: Parser String
seqP    :: Parser a -> Parser b -> Parser b
(><)    :: Parser a -> Parser b -> Parser (a,b)
repeatP :: Parser a -> Parser [a]

theToken tok = match (show tok) (\tok0 -> if tok==tok0 then Just tok else Nothing)
ident = match "<ident>" (\tok0 -> case tok0 of
   ALPHA x -> Just x
   _ -> Nothing)
p1 `seqP` p2 = p1 `thenP` const p2
(p1 >< p2) = p1 `thenP` \x -> p2 `thenP` \y -> returnP (x,y)
repeatP p = (p >< repeatP p `thenP` (returnP . uncurry (:))) `altP` returnP □
```

Designing the Standard Haskell Libraries (Position Paper)

Alastair Reid and John Peterson
Department of Computer Science, Yale University,
P.O. Box 208285, New Haven, CT 06520, USA.
Electronic mail: {reid-alastair,peterson-john}@cs.yale.edu

June 4, 1995

Abstract

Five years after the first Haskell report was published, the Haskell language continues to grow and mature. After five years experience of Haskell programming, we wish to both expand and simplify the Haskell language. Over the years, many Haskell libraries have been developed. The Haskell Committee is expanding the language by standardising a set of libraries to add to the definition of Haskell. Another goal is to simplify Haskell by moving parts of the prelude, a built-in set of types and functions implicitly a part of every Haskell program, into libraries where they can be loaded on demand, This document describes the issues involved in the design of the Haskell libraries and summarises the library modules being considered.

1 Motivation

In the five years since the Haskell programming language was created, Haskell programmers have developed libraries providing many useful functions and datatypes. Each implementation of Haskell currently distributes home grown libraries. Using these libraries speeds development but decreases portability because these libraries are not available on all platforms. We believe that many of the functions and types in these libraries should become a standardised part of Haskell 1.3 (with further libraries being added in later revisions). Standardising these libraries will:

1. Increase the power of the language by providing a much greater level of basic functionality in the standard.
2. Improve the portability of programs by eliminating the need for non-standard libraries.
3. Avoid the splintering of Haskell into different dialects where programmers familiar with the HBC libraries (say) would not be able to understand a program written using the GHC libraries.

The major argument against standardising libraries is that doing so increases the size of language — both the amount new programmers must learn and the size of implementations. We feel that the increased utility of the language outweighs this concern.

The Haskell language has what is essentially a built-in library called the standard prelude. The prelude is special only in that it is implicitly imported into every program. Moving the infrequently-used components of the prelude into libraries has two advantages: it shrinks the core language, making the essential components of Haskell more apparent; and it frees up more of the namespace for the user.

This document describes the issues that arise in turning the existing libraries into a concrete library proposal and briefly summarises each library. This document does not discuss changes to the libraries or prelude which are related to the I/O proposal; these are described in Gordon and Hammond's tutorial paper [3]. Detailed definitions of the libraries are supplied in a companion document [11].

Acknowledgements

We are grateful to members of the Yale, Glasgow and Chalmers functional programming groups (especially Sandra Loosemore, Dan Rabin, Will Partain, Jim Mattson, Andy Gill, Kevin Hammond and Kent Karlsson), Mark Jones, Ian Poole, Nick North, Paul Otto and the other members of the Haskell 1.3 committee for detailed comments on the design criteria and the design of the libraries.

We are especially grateful to the GRASP/AQUA team at Glasgow and to Lennart Augustsson at Chalmers for providing the basis on which much of the library design is built.

2 Design Issues

The main questions to be answered when designing library modules are:

What should be included?
What should go in the libraries and what should go in the prelude?
What should go in each module?
What classes should each new type be an instance of?
What should the type and name of each function be?
How should library functions be defined in a standard?
How do libraries interact with other aspects of Haskell?

We address each of these questions in turn.

2.1 What should be included?

To be included in the standard libraries, an entity (type, type-class or function) must satisfy two criteria:

1. It must be clear what the interface should look like — it is much harder to remove or correct a feature once it has been included in the language standard.

2. The entity must be useful to a "significant number" of programmers.

Since the standard libraries draw heavily on existing libraries, we have some confidence that the interfaces have been tested in real programs.

Entities which cannot be implemented efficiently in standard Haskell because they require special support in the compiler or runtime system deserve special consideration since users have no portable alternative.

Since the primary goal of having standard libraries is to improve portability, Haskell implementations are required to provide all of the standard libraries and are not permitted to add, modify, or omit entities. Implementors are encouraged to provide optimised versions of library functions *provided that the optimised version has the same external behaviour (type, strictness, error conditions, etc) as in the specification.*

2.2 What should go in the libraries and what should go in the prelude?

The *only* operational difference between entities defined in the prelude and those defined in a library is that the prelude is automatically imported into every Haskell module whereas libraries must be explicitly imported. This division into prelude and libraries does not imply that the libraries are optional (they're not) or that the prelude cannot import library modules (it can). On the other hand, entities in the prelude can be considered "more essential" to the language. Students of Haskell would be expected to learn about the prelude before looking into the libraries.

It is slightly more convenient to use entities from the standard prelude than from a library. But, each entity placed in the prelude "steals" a potentially useful name from programmers. To avoid name clashes, programmers must give their own entities different names, or use hiding or renaming.

In order to justify "stealing a name" from the user, each entity in the prelude must satisfy one or more of the following criteria:

1. It is very heavily used by all programmers. For example, the existing functions `map` and `show` are so heavily used that programmers are unlikely to use the name for any other purpose.

2. It occurs in introductory functional programming courses/textbooks. For example, `interact` is not heavily used in large programs but including it in the prelude avoids or delays the need to teach students about Haskell's module system.

Experience with Haskell 1.2 suggests that arrays, rational numbers and complex numbers are not used heavily enough to justify their inclusion in the prelude. Therefore `PreludeArray`, `PreludeRatio` and `PreludeComplex` will be libraries in Haskell 1.3. Similarly, the functions `ord`, `chr`, `isControl`, `isPrint`, etc. in the module `Prelude` are rarely used and will be moved to module `LibCharType`.

2.3 What should go in each module?

Each library module comprises one of:

1. A single (abstract) data type. (Or a family of types and corresponding type class in the case of `LibWord`.)
2. A single type class.
3. A set of closely related utility functions.

   For example, operations on lists are divided into modules `LibLength`, `LibDuplicates`, `LibScan`, `LibSubsequences`, etc. rather then being grouped into a single module. Our goal is to keep libraries small and self-contained so that programmers can import those functions they need without importing many functions they don't need.

2.4 What classes should each new type be an instance of?

In Haskell 1.2, an instance of a class must be defined in either the module that defines the class or in the module that defines the instance. This rule makes it impossible for the programmer to add an instance if it has been omitted from the prelude or libraries. Therefore care must be taken to define every possible instance of every possible class to avoid leaving the programmer high and dry.

At the time of writing, it seems likely that this rule will be relaxed in Haskell 1.3 to allow a given instance to be defined anywhere in the program — provided there is at most one instance. Nevertheless, it is important to provide every possible instance for all abstract data types — and most reasonable instances for all other types.

The question of where to define an instance is also important. If instances are defined in the modules that defines the classes, importing a class might cause a large amount of unwanted code (associated with types that the programmer is not using) to come into scope. If instances are defined in the modules that define the types, importing a type might cause a large number of unwanted code (associated with classes the programmer is not using) to come into scope. Both seem to result in a very cluttered name space and large compiled programs. We don't have a good solution at the moment.

2.5 What should the type and name of each function be?

The primary goal in choosing names is that it should be possible to guess the purpose of a function from its name and type signature. In some cases it may be appropriate to change the names of existing prelude functions to achieve this goal (e.g. a better name for `null` would be `isEmpty`).

Secondary considerations include:

1. Consistency with the existing Haskell prelude.

   For example, modules `LibSet`, `LibBag` and `LibFiniteMap` all provide a function analogous to the prelude function `filter` to "select" values from a collection. We use the names `filterSet`, `filterBag` and `filterFM` for these functions.

2. Consistency between different libraries.

   For example, modules `LibSet`, `LibBag` and `LibFiniteMap` all provide a function to combine sets, bags or datatypes (respectively). We use similar names (`unionSet`, `unionBag` and `unionFM`) for all three functions.

   At the time of writing, it seems likely that Haskell 1.3 will provide a form of "qualified names" allowing one to import several entities with the same name and using the module name as a qualifier to resolve any ambiguity. If this proposal is adopted, both the `Lib` prefixes on the module names and the type suffixes on the function names would be dropped — the functions being called `Set.union`, `Bag.union` and `FiniteMap.union` respectively.

3. Consistency with the existing Haskell naming conventions.

   For example, identifiers formed by the concatenation of several words use capitalisation rather than underscores to separate the words.

4. We try to avoid using a name if a programmer might reasonably use the name for some other purpose.

In a language that encourages use of partial application and allows any binary function to be used as an infix operator, it is important to consider possible uses of a function when choosing the order of arguments. For example, a function which modifies an object of type `T` should take this object as the last argument. Thus `add` would have type `a -> Set a -> Set a` instead of `Set a -> a -> Set a`.

2.6 How should library functions be defined in a standard?

With the exception of primitive arithmetic and I/O-related functions, all functions in the Haskell 1.2 prelude are described in English in the body of the report and defined in Haskell in an appendix. Providing a Haskell definition avoids ambiguity but can be very verbose and hard to understand (see, for example, `PreludeText`).

For some functions, we might wish to be deliberately ambiguous: all a programmer needs to know about a sort function is whether it is stable and for which inputs it behaves efficiently — details about the choice of algorithm are best left to the implementor. For these functions, it is more appropriate to provide an English description of the function backed up by mathematical identities, error conditions and strictness properties as appropriate.

For all other functions (i.e., those simple enough that they are best specified in Haskell), the required semantics is precisely that of the definition — implementors are *not* free to

change the semantics to improve performance.[1]

Some types such bitsets and random states we wish to leave the implementor free to choose an efficient representation but wish to constrain the behaviour sufficiently to guarantee portability. In these cases, a reference implementation is provided in Haskell but the representation is not exported from the defining module.

Dealing with the strictness properties of library functions is a particular problem. In many cases, the strictness of a function depends on the order in which tests for errors or special conditions are made. For some functions, laziness is crucial to the program; for others, it makes little difference. The standard can either be very precise about strictness or it can allow implementations to choose whatever strictness properties lead to the best implementation. We propose to explicitly mark functions for which the implementation is free to alter the strictness properties. Most of the others can be defined in Haskell, which precisely defines the strictness. To avoid introducing subtle portability problems, we plan to keep the number of functions with undefined strictness properties as small as possible.

As in the existing prelude, implementations are free to alter calls to the `error` function in library functions. Error messages may be changed freely to make them more useful. Library functions should detect errors as early as possible and report them clearly.

The libraries introduce a number of new types which are similar to existing types (e.g. `PackedStrings` are similar to `Strings`, `FiniteMaps` to association lists and arrays, `Sets` to lists). The relationship between two types can often be described by a pair of functions $(f :: U \to V, g :: V \to U)$ such that $g \circ f = id$. By abuse, we call such pairs "retraction pairs" and write $(f, g) : U \leftrightarrow V$. (The complete definition of "retraction pairs" would require that $f \circ g \sqsubseteq id$ for an appropriate choice of domain.) When a retraction pair exists, it is natural to use the pair to define the semantics of functions over the new types in terms of corresponding functions over the old types. For example, the retraction pair for packed strings (`unpackPS`, `packString`) : `PackedString` $\leftrightarrow$ `String` can be used to define the functions

```
headPS :: PackedString -> Char
headPS = head . unpackPS

tailPS :: PackedString -> PackedString
tailPS = packString . tail . unpackPS

nullPS :: PackedString -> Bool
nullPS = null . unpackPS
```

When accompanied by a definition of the composition `unpackPS . packString`, this completely specifies the required behaviour of these functions — though one would hope for more efficient implementations!

[1] Many Haskell compilers break this rule for prelude functions. Since this reduces portability, we recommend that they provide a compiler option to force the use of a correct (but possibly slower) implementation — as far as is possible.

2.7 How do libraries interact with other aspects of Haskell?

Although the Haskell report is silent on exactly how the components of a program are gathered for compilation, we expect that the user should not have to do anything special to specify the location of modules in the standard Haskell library. Simply mentioning a library name in an `import` declaration ought to be sufficient to link the appropriate library into the Haskell program.

Libraries may define derivable classes. The names of such classes must be explicitly imported into the modules that define types which derive them. It is the responsibility of the compiler to correctly expand a `deriving` clause involving such a class.

It is also possible that the compiler-generated code for a `deriving` clause may reference entities defined in libraries. It it is not necessary for the programmer to import these implicitly-referenced entities, although the compiler must arrange for them to be linked into the resulting program.

3 An Overview of the Proposed Libraries

This section summarises those libraries which will be included in Haskell 1.3. (A complete definition may be found in document [11].) We omit `LibCharType` which provides character operations removed from `Prelude`, and `LibArray`, `LibComplex` and `LibRatio` which are just old prelude modules turned into libraries.

3.1 Non-overloaded prelude functions

Functions such as `elem` use the `Eq` class to supply the `==` operation. There are situations in which the operation defined by overloading is not appropriate. For example, one might wish to use a case-insensitive comparison when operating on strings. It is straightforward to define versions of these functions which are not overloaded (but are polymorphic) by adding an extra argument which provides an explicit equality or comparison predicate. For example, we have `nubBy :: (a -> a -> Bool) -> [a] -> [a]`. While it is trivial to define overloaded functions in terms of non-overloaded ones, the reverse is not possible.

The prelude provides several functions which are overloaded with respect to `Eq` and `Ord`: `nub`, `elem`, `notElem`, `min`, `max`, `maximum`, `minimum` and `(\\)`. Rather than creating a module containing a random assortment of such functions, we place the non-overloaded version in the same module as the original definition. The overloaded version can be defined using the new function; for example, `nub` can be defined by `nub = nubBy (==)`.

3.2 Packed Strings

Haskell represents strings by lists of characters. While this interacts well with lazy evaluation and allows many prelude functions (such as `map` and `length`) to be used on strings, typical Haskell implementations consume 20–40 bytes per character to represent such strings.

The module `LibPackedString` provides a new type `PackedString` which is evaluated more strictly to allow a more compact representation (as low as 1 byte per character plus a small constant overhead).

The retraction pair (`unpackPS`, `packString`) : `PackedString` $\leftrightarrow$ `String` is not quite an isomorphism since `packString` completely evaluates its argument. These functions are used to define `PackedString` versions of the prelude constructors `[]` and `(:)` and the prelude functions `head`, `tail`, `init`, `last`, `null`, `length`, `append`, `map`, `filter`, `foldl`, `foldr`, `take`, `drop`, `splitAt`, `takeWhile`, `dropWhile`, `span`, `break`, `lines`, `words`, `reverse`, `concat`, `elem` and `(!!)`.

This module is based on the `PackedString` module distributed with GHC.

3.3 List Operations

As one of the most heavily used data structures in Haskell, it is not surprising that the last five years use has produced a host of useful new functions over lists.

`LibSort`

This module provides a *stable* sorting function `sort :: Ord a => [a] -> [a]`. (`sortBy` is also provided.)

`LibDuplicates`

This module provides functions for manipulating lists with duplicate values: `group :: Eq a => [a] -> [[a]]` and `uniq :: Eq a => [a] -> [a]` which group together adjacent equal elements in a list and eliminate adjacent equal elements. (The function `LibSort.sort` can be used to bring duplicates together.) (`groupBy` and `uniqBy` are also provided.)

`LibLength`

This module provides functions such as `lengthLe, lengthEq :: [a] -> Int -> Bool` to test the length of a list without evaluating the entire list.

`LibScans`

This module provides unidirectional and bidirectional generalisations of `fold` and `scan` based on functions provided in HBC's ListUtils module, GHC's Utils module and O'Donnell's paper on bidirectional fold and scan [9].

`LibSubsequences`

This module provides functions to generate the list of all subsequences, prefixes, suffixes or permutations of a list and to test whether one list is a subsequence, prefix, suffix or permutation of another list. These are based on functions defined by Bird and Wadler [2].

3.4 Collections

Lists are very heavily used in Haskell programs. Sadly, lists can be quite inefficient (in time) for storing large collections of data and it is possible to do significantly better using data structures based on binary trees or hash tables.

The modules `LibBag`, `LibSet`, `LibFiniteMap` and `LibHashTable` define types `Bag`, `Set`, `FiniteMap` and `HashTable`, retraction pairs relating them to lists or association lists and functions over these types (mostly based on prelude functions over lists and arrays).

`Bag` α

The type `Bag` α is isomorphic to $[\alpha]$ but provides constant time appending and concatenation functions and logarithmic time `head` and `last` functions.

The retraction pair for bags (`toList`, `fromList`) : `Bag` $\alpha \leftrightarrow [\alpha]$ forms an isomorphism. That is

```
toList . fromList = id
```

`Set` α

The type `Set` represents (ordered) collections with no duplicates.

The retraction pair for sets (`toList`, `fromList`) : `Ord` $\alpha \Rightarrow$ `Set` $\alpha \leftrightarrow [\alpha]$ satisfies the identity

```
toList . fromList = uniq . sort
```

`FiniteMap` α β and `HashTable` α β

The types `FiniteMap` α β and `HashTable` α β represent lookup tables with keys of type α and elements of type β. `FiniteMaps` behave like balanced binary trees (logarithmic access time and insertion time) and `HashTables` behave like (functional) hash tables (near-constant access time and linear insertion time).

The retraction pair for finite maps (`toList`, `fromList`) : `Ord` $\alpha \Rightarrow$ `FiniteMap` α $\beta \leftrightarrow$ $[(\alpha, \beta)]$ satisfies the identity

```
toList . fromList = uniqBy eq . sortBy cmp
 where
  (x,_) `cmp` (y,_) = x <= y
  (x,_) `eq` (y,_) = x == y
```

Hash tables require a new type class `Hashable`. The following laws are satisfied by the retraction pair for hash tables (`toList`, `fromList`) : `Hashable` $\alpha \Rightarrow$ `HashTable` α $\beta \leftrightarrow$ $[(\alpha, \beta)]$

```
map fst xs `isPermutationOf` map fst (toList (fromList xs))
lookup xs = lookup (toList (fromList xs))
```

All four modules provide functions corresponding to the prelude constructors [] and (:) and of the prelude functions ++, \\, length, genericLength, map, partition, filter, foldl and foldr. LibSet and LibBag also provide versions of elem and notElem; LibBag provides versions of head, tail, init, last, (!!) and reverse; and LibFiniteMap and LibHashTable provides versions of indices, elems, accum, (//), (!), amap and ixmap. (All functions are defined using the retraction pairs.)

The module LibHash is provided to help support LibHashTable. It provides a new abstract type Hash, a new type class Hashable defining a method hash :: Hashable a => a -> Hash and instances for Haskell's primitive types (e.g., Int, Integer). Instances of Hashable may be derived.

Modules LibBag, LibSet and LibFiniteMap are based on modules distributed with GHC; LibHash is loosely based on a module distributed with HBC (the type Hash is just Int in the HBC version.)

3.5 Monads

Since their first use in pure functional programming [13], monads have revolutionised the way programmers perform input/output, update-in-place, parsing, exception handling and many other tasks.

If constructor classes [6] are added to Haskell 1.3, it would be possible to define constructor classes representing monads, monads with a zero element and monads with a choice operator. Such classes could be defined in a library but are sufficiently important to justify their addition to the prelude. Instances would include lists, Maybe, Either, IO and Parser. There are a number of useful monadic functions which can be defined using these operations. The current interface is based on monads used within GHC and on examples distributed with Gofer.

3.6 Mutable Structures

Peyton Jones, Wadler and Launchbury [7, 8] describe how monads may be used to provide mutable variables and mutable arrays without losing referential transparency.

The module LibMutable provides both mutable variables and mutable arrays. The major unresolved question is whether the operations to create, read and write mutable structures should be part of the IO monad, part of a state thread monad or whether the IO monad should be an instance of state thread monads as in Peyton Jones and Launchbury's elegant lazy state threads paper [8]. The problem is that their approach requires the addition of a new language construct runST with special type-checking rules. It is not yet clear whether the extra power justifies complicating the language. This module is based on the PreludeGlaST module distributed with GHC.

3.7 Printing and Parsing

The prelude provides the `Text` class for printing and parsing values. Derived methods have the desirable property that `read . show = id` (for non-functions and ignoring strictness). However, the output from `show` can be rather ugly and it is awkward to construct good parsers using `read`.

The module `LibPretty` provides a new abstract type `Pretty` representing a pretty-printed block of text and functions for combining values of type `Pretty` in various ways useful when printing programs and data structures. The current interface is based on Hughes' pretty-printing library as distributed with HBC and GHC.

The module `LibParse` provides a new abstract type `Parser` α β of backtracking recursive-descent parsers which consume a token stream of type α and produces "parse trees" of type β. The current interface is based on Hutton's parsing library [5] as distributed with HBC.

3.8 Binary Files

The class `Text` provides a limited form of persistence: most built in and user-defined types can be printed to text files and subsequently read back in. However, the process of converting values to strings and using a backtracking lexer and parser to read them back in is notoriously inefficient. Haskell 1.2 provided the `Bin` datatype and the `Binary` type class for efficiently writing values to files in a more concise (implementation-specific) manner.

This feature of Haskell 1.2 was essential to some programs but rarely used and so is being moved into a library. At the same time, the adoption of monadic I/O makes it possible to provide operations to read and write values to a binary file directly — eliminating the need to create an intermediate `Bin` value.

To ensure portability of generated files, the specification of this module should define the precise external representation for each datatype in class `Binary`. Some care is required to avoid overly restricting the possible range of certain datatypes such as `Int` and `Float`.

Instances of `Binary` may be derived. The current interface is based on the Native module distributed with HBC with extensions based on the current Yale implementation.

3.9 Random Numbers and Splittable Supplies

While functional programming languages are valued because their results are deterministic and easy to predict, many kinds of programs such as simulations and games need to appear non-deterministic — they need a supply of random numbers.

The module `LibRandom` provides a new abstract type: `RandomState` together with a function `nextRandomState :: RandomState -> RandomState`; and a new type class `Random` which provides a method `fromRandomState :: Random a => RandomState -> a`. There are also functions for initialising random states, and generating lists of random values and `Text` and `Binary` instances for reading and writing random states to files.

Although the type `RandomState` is abstract, the definition will precisely specify the algorithm used to generate random numbers to ensure consistent results across all implementations.

The current interface and semantics is loosely based on the random number operations in Common Lisp.

The module `LibSupply` provides a new abstract type `Supply` α which represents a splittable supply of values of type α. This module is included to support supplies of random numbers but has also proved useful within compilers. The current interface is based on Augustsson, Rittri and Synek's splittable supply library [1] as distributed with HBC.

The price paid for an efficient splittable supply implementation is that programs using the splittable supply module might produce different results when compiled with different compilers, with different optimisation options or even after small semantics-preserving changes to the program. In practice, this does not present any problems. For example, in a type-checker, it is important to have an efficient supply of distinct type variables but it is irrelevant where each type variable is used.

3.10 BitSets

Bitwise operations are useful for two purposes: 1) they provide fast, compact operations on sets of small integers; and 2) they are useful in communicating with hardware devices, network protocols, etc.

The module `LibBitSet` provides types `Word8`, `Word16`, `Word32`, `Word64`, `Word128` and `Word` (unbounded words) and a type class `BitSet` which provides the usual bit-manipulation operations (including both arithmetic and logical shifting operations). The new types are instances of `Integral` (and all superclasses); negative numbers are interpreted as though they are represented in two's-complement notation. The current interface is loosely based on the Common Lisp logical operations [12] and the `Word` library distributed with HBC.

3.11 Future Work

There is a considerable amount of work ahead designing the exact interface to these modules, defining the precise semantics of these operations and documenting the resulting modules.

We are also considering libraries to support matrix operations, regular expressions, Hughes' lazy memo functions [4], Johnsson's lazy arrays, the language independent arithmetic standard [10] and the X11 graphics protocol.

References

[1] L Augustsson, M Rittri, and D Synek. On generating unique names. *Journal of Functional Programming*, 4(1):117–123, January 1994.

[2] RS Bird and PL Wadler. *Introduction to Functional Programming.* Prentice-Hall, 1988.

[3] AD Gordon and K Hammond. Monadic I/O in Haskell. In *Proceedings of Haskell Workshop*, June 1995.

[4] John Hughes. Lazy memo-functions. In *Functional Programming and Computer Architecture*, pages 129–146, September 1985.

[5] G Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, 2(3):323–343, July 1992.

[6] MP Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. In *Functional Programming and Computer Architecture*, 1993.

[7] SL Peyton Jones and PL Wadler. Imperative functional programming. In *Principles of Programming Languages*, pages 71–84. ACM, January 1993.

[8] J Launchbury and SL Peyton Jones. Lazy functional state threads. In *Programming Languages Design and Implementation*, 1994.

[9] JT O'Donnell. Bidirectional fold and scan. In *Draft Proceedings of Glasgow Functional Programming Workshop*, pages XX1 - XX6. Glasgow Functional Programming Group, July 1993.

[10] M Payne, C Schaffert, and BA Wichmann. The language compatible arithmetic standard. *ACM SIGPLAN Notices*, 25(1):59–86, January 1990.

[11] AD Reid and JC Peterson. A proposal for the standard Haskell libraries. 1995. In preparation for distribution at Haskell Workshop.

[12] GL Steele. *Common Lisp — The Language.* Digital Press, 2nd edition, 1994.

[13] PL Wadler. Comprehending monads. In *Proc ACM Conference on Lisp and Functional Programming, Nice.* ACM, June 1990.

Adding Records to Haskell

John Peterson and Alastair Reid
Department of Computer Science, Yale University,
P.O. Box 208285, New Haven, CT 06520, USA.
Electronic mail: {peterson-john,reid-alastair}@cs.yale.edu

June 5, 1995

Abstract

The Haskell programming language has a very simple yet elegant view of data structures. Unfortunately, this minimalist approach to data structures, in particular record-like structures, presents serious software engineering problems. We have implemented an extension to standard Haskell which provides record-like structures in addition to ordinary algebraic data types. Our extension provides named fields in data structures, default field values, field update functions, detection of uninitialized slots and multiple inheritance. Our major design goal was to supply as much functionality as possible without changing any of the basic components of the Haskell language (in particular, we avoided further complication of the type system). The purpose of this paper is not to advocate this specific extension to Haskell, but to examine the basic engineering issues associated with records; describe our experiences with the implementation and use of one particular proposal; and consider alternative approaches (some of which have been used in other languages).

1 Introduction

The Haskell language [2] includes only the most basic support for a fundamental programming language feature: the record type. In the most general sense of the term, a record simply groups a heterogeneous collection of objects into a single value. There are many different manifestations of record-like features in programming languages, including tuples, structures, and objects. While the algebraic data types found in Haskell have the necessary functionality to build record data structures, Haskell lacks many desirable features found in other languages for dealing with complex data objects.

This paper describes our implementation of record types in the Yale Haskell system. The purpose of this experiment is not to advocate any specific implementation of records in Haskell, but to fully explore one possible approach to this problem and to gain practical experience with the problem of integrating records with the Haskell programming style. After presenting our implementation, we compare our system of records to those found in other languages and discuss alternatives to our design.

Before proceeding, we will clarify our terminology. We use the term *record* in only the most general sense. The components of records are *fields*. Within the context of our specific proposal, we use the terms *structure* and *slot* to denote our particular implementation of records and fields (respectively).

The issues of concern here are not so much in the fundamental language semantics, but instead are matters of engineering. From a software engineering standpoint, the record structures provided by a programming language benefit from the following properties:

- Expandability. Adding a new field to a record should not require modification of code which references old fields. It should be possible to new fields silently without changing existing code.
- Reusability. A record should be able to include (inherit) other records; operations which apply to the included records should also apply the including record.
- Efficiency. Basic record operations must be extremely efficient; there must be no hidden performance costs.
- Privacy. The program must be able to hide the internal details of a record.

Along with these engineering issues, we have one further goal: to keep our system as much in the spirit of standard Haskell as possible.

The basic features of our proposal are:

- The semantics are entirely defined via a translation to standard Haskell. No modifications are required in the Haskell type system.
- Slots may be accessed via pattern matching or by function application.
- Slots may be (functionally) updated.
- Default values may be provided for slots.
- Uninitialized slots can be detected by the programmer.
- Special syntax is used for creating, updating, coercing, etc. This avoids generating new names for these operations (as is done in Common Lisp [6]).
- Explicit declarations are required for all record types. This avoids the efficiency and type-inference problems associated with more general record types and produces more accurate messages when type errors occur.
- Structures may be polymorphic.
- Multiple inheritance is allowed. Inheritance is implemented using Haskell's type class mechanism; structure operations and user-defined functions are overloaded to allow them to apply to any structures defining appropriate fields. Coercion functions are provided to move up and down the inheritance graph.

2 Data structuring in standard Haskell

Before presenting our proposal, we explore what can be done in standard Haskell. This both illustrates the need for improvement and provides the basis for describing the semantics of our proposal.

For example the following datatype which is used to represent named entities within the Yale Haskell compiler.[1]

```
data Definition =
     MkDef
       String          -- name
       String          -- module in which it is defined
       String          -- unit in which it is defined
       Bool            -- is it exported?
       Bool            -- is it a PreludeCore symbol?
       Bool            -- is it a Prelude symbol?
       Bool            -- is it created by an interface?
       Bool            -- is it ''made up'' by the compiler?
       (Maybe SrcLoc) -- where it was defined.
```

This datatype is hard to use reliably. There are several fields of the same type — the type system is not able to detect simple errors such as accidentally swapping the fourth and fifth fields. Such problems are very difficult to spot when fields are identified only by their position with respect to a constructor. It is also hard to maintain: adding an extra field to this definition requires changes to every use of the constructor `MkDef` (i.e. taking `Definitions` apart in patterns and constructing `Definitions` in expressions.)

The usual solution to the problem of reliably handling many fields is to define "access functions" for updating and selecting each field of the record. For this example, we must define 18 different access functions — one to extract each slot and one to update each slot:

```
getName, getModule, getUnit :: Definition -> String
getName   (MkDef nm _ _ _ _ _ _ _ _ _)   = nm
getModule (MkDef _ mod _ _ _ _ _ _ _)  = mod
getUnit   (MkDef _ _ unit _ _ _ _ _ _) = unit
...

setName, setModule, setUnit :: String -> Definition -> Definition
setName nm (MkDef _ mod unit isEx isCore isPrel isIface isInternal loc)
  = (MkDef nm mod unit isEx isCore isPrel isIface isInternal loc)
setModule mod (MkDef nm _ unit isEx isCore isPrel isIface isInternal loc)
  = (MkDef nm mod unit isEx isCore isPrel isIface isInternal loc)
setUnit unit (MkDef nm mod _ isEx isCore isPrel isIface isInternal loc)
  = (MkDef nm mod unit isEx isCore isPrel isIface isInternal loc)
...
```

[1]The Yale Haskell compiler is written in Lisp; this example is obtained by translating from Lisp to Haskell. Similar examples occur in the Glasgow Haskell compiler — which *is* written in Haskell.

Using these access functions instead of referencing the constructor `MkDef` directly results in more readable code and simplifies the task of adding new fields to a record. However, the reader will appreciate that creation of these access functions is a somewhat tedious and error-prone task.

A further problem with this approach is that it is no longer possible to use pattern matching to extract components of records. This makes programs more verbose.

3 Syntactic Support for Records

The core of our proposal is to provide special syntax for defining structure types, accessing slots, and initializing structures. The semantics of our proposal is defined as a translation into code like that given in the previous section.

The additions to Haskell syntax rules (appendix B of [2]) are as follows:

3.1 Structure declarations

topdecl	→	`structure` [`~`] *simple* `where` { *structbody* [;] } [`deriving` (*tyclses*)]
simple	→	*tycon tyvar*$_1$ ... *tyvar*$_k$
structbody	→	*structsigns* [; *valdefs*]
structsigns	→	*structsign*$_1$; ... ; *structsign*$_n$
structsign	→	*vars* :: [*context* =>] *type*

Using this syntax, the datatype and access functions in section 2 can be more concisely defined by

```
structure ~Definition where
  name, moduleName, unit        :: String
  isExported, isCore, isPrelude :: Bool
  fromInterface, isInternalDef  :: Bool
  definedIn                     :: Maybe SourceLoc
```

(The "twiddle" is related to inheritance and is described in section 4.1.)

The selector functions have exactly the same name as the slot they extract; for example, the following function prints the original name of a definition:

```
showDefName :: Definition -> ShowS
showDefName d
   = showString (moduleName d) . showChar '.' . showString (name d)
```

Translation: The declaration

```
structure ~S t1...tk where
  v1 :: u1; ... ; vm :: um
  vi1 = init1; ... ; vin = initn
```

is equivalent to the type declaration and function declarations

```
data S t1...tk = MkS u1 ... um
v1 :: S t1...tk -> u1
v1 (MkS x1 ... xm) = x1
⋮
vm :: S t1...tk -> um
vm (MkS x1 ... xm) = xm
```

(The meaning of the default values (v_{i1} = $init_1$; ... ; v_{in} = $init_{in}$) is defined below; the meaning of omitting the "twiddle" (~) is defined in section 4.1.)

Note: Although we define the semantics of our system by translation into standard Haskell, the constructor MkS used in this translation is not made directly available to the programmer.

3.2 Pattern Matching

apat → ($spat_1$, ... , $spat_n$) (structure pattern, $n \geq 1$)
spat → *var* = *pat*

An alternative way of extracting slots is by pattern matching. Structure patterns consist of a list of pairs of slot-names and patterns. For example, the function `showDefName` could be written as:

```
showDefName :: Definition -> ShowS
showDefName (moduleName = m, name = nm)
  = showString m . showChar '.' . showString nm
```

The order in which slot names are listed does not matter. Pattern matching proceeds left to right as in all other patterns.

Translation: The expression `case` e_0 `of (`$s1$ `=` p_1`, ... `sn `=` p_n`) -> `e`; _ ->` e' , for slot names $s1, \ldots sn$, is equivalent to:

```
let { y = e' } in
case e0 of { MkS x1 ... xk ->
case xs1 of { p1 -> ... case xsn of { pn -> e ; _ -> y } ...
              _ -> y
}}
```

where y, $x_1 \ldots x_k$ are new variables and x_s is the value of the slot named s.

3.3 Updates

aexp	→	(*var* =)	(*update section*)
	|	(upd_1 , ... , upd_n)	(*update function*, $n \geq 1$)
upd	→	*var* = *exp*	

For each slot $s :: t$ of a structure S, the update section (s=) is a function of type $t \rightarrow S \rightarrow S$ which copies the structure updating the value of the slot s. The value to be placed in the slot can be placed inside the parenthesis, as in `(name = "foo")`. More than one slot can be updated at once, as in `(name = ".", moduleName = "Prelude")`.

Translation: Given the structure declaration
`structure ~`S $t_1 \ldots t_k$ `where`
v_1 :: u_1; ... ; v_m :: u_m
v_{i1} = $init_1$; ... ; v_{in} = $init_n$
the notation (v_i =) is equivalent to:
(\ x (MkS x_1 ... x_i ... x_m) -> (MkS x_1 ... x ... x_m))
and the notation (v_{i1} = e_1, ...v_{in} = e_n) is equivalent to:

(\ s -> (v_{i1}=) e_1 (... (v_{in}=) e_n s ...))

The order in which slot names are listed does not matter; but it is a static error to use the same slot name more than once.

3.4 Structure Creation

There is no special syntax for structure creation. The structure name is used as a modified data constructor: instead of being applied to the component values, this constructor applies an update function to an initial value constructed from the defaults specified in the structure declaration.

For example, the function `mkCoreDef` creates a `PreludeCore` definition. The list of slot names and values is an update function as defined in the previous section and `Definition` is a function which applies the update function to an "empty" structure in which each slot is undefined (there are no default slot values declared in this example).

```
mkCoreDef :: String -> SourceLoc -> Definition
mkCoreDef nm src = Definition (
                    name = nm,
                    moduleName = "PreludeCore",
                    isExported = True,
                    isCore = True,
                    isPrelude = True,
                    fromInterface = False,
                    definedIn = src
                    )
```

The syntax for declaring structures allows default values to be specified for some of the slots. A straightforward approach would require the default value of each slot to have the same type as the slot. For example, one might add the following default values to the structure declaration.

```
...
isExported = False
isCore = False
isPrelude = False
fromInterface = False
definedIn = Nothing
```

However, by making the default a function mapping the structure being defined onto a slot value it becomes possible for default values to depend on the values of other slots — particularly those of explicitly-initialized slots. For example, the values of the slots `isCore` and `isPrelude` can be made to depend on the value of the `moduleName` slot.

```
...
isExported self = False
isCore (moduleName = mod) = mod == "PreludeCore"
isPrelude (moduleName = mod) = take 7 mod == "Prelude"
fromInterface self = False
definedIn self = Nothing
```

The implementation of this style of default argument is somewhat subtle: we use a recursion to allow explicitly initialized slots to override default values and to allow default values to depend on other slots in the same structure.

Translation:
For a structure type constructor S, the occurrence of the type constructor S in an expression is equivalent to the function

```
\init -> let s = init ((v1 = init1 s, ..., vn = initn s) (MkS ⊥...⊥)
          in s)
```

where the default values for variables $v_1, \ldots, v_n$ are $init_1, \ldots, init_n$ (respectively).

It is a static error to provide more than one default value for a slot. Uninitialized slots with no default are bound to error calls.

Strictness annotations in data type definitions cause problems with initialization: an uninitialized structure slot would immediately cause a program error. Our solution is that strict slots must have a default value and that default value should have the same type as the slot (rather than being a function whose argument is the structure being created). This constant is used instead of $\perp$ in the above translation.

3.5 Uninitialized Slots

Slots which have no default value may remain uninitialized by structure creation. While accessing such slots results in a runtime error, it is sometimes useful to test whether a slot is initialized without actually referencing its value. It is, of course, possible to avoid this by adopting the convention that every slot must have a default value. On the other hand, by allowing uninitialized slots to be detectable, a robust derived `Text` instance for structures can simply skip over uninitialized slots instead of crashing when attempting to access such a slot.

The changes to the translation are straightforward but tedious: the datatype

`data` $S\ t_1 \ldots t_k$ `=` $MkS\ u_1\ \ldots\ u_m$

is changed to

`data` $S\ t_1 \ldots t_k$ `=` MkS `(Maybe` u_1`)` `...` `(Maybe` u_m`)`

The definition of selector functions and update sections are modified to accommodate this change

v_i `(`$MkS\ x_1$ `...` `(Just` x_i`)` `...` x_m`)` `=` x_i
`(`v_i `=)` `=` `\ x (`$MkS\ x_1$ `...` x_i `...` x_m`)` `->` `(`$S\ x_1$ `...` `(Just` x`)` `...` x_m`)`

and (in the absence of an explicit default) the default value of every slot is changed from $\perp$ to `Nothing`.

Translation: Given the structure declaration
`structure ~`$S\ t_1 \ldots t_k$ `where`
v_1 `::` u_1`;` `...` `;` v_m `::` u_m
v_{i1} `=` $init_1$`;` `...` `;` v_{in} `=` $init_{in}$
the notation `(=` v_i`)` is equivalent to:
`(\` $MkS\ x_1 \ldots x_i \ldots x_m$`) ->`
`case` x_i `of { Just _ -> True; Nothing -> False })`

This translation is rather inefficient — imposing an overhead on creation, selection and updates. Fortunately, it is easy to detect undefined slots without an explicit `Maybe` datatype in the representation. To produce meaningful error messages, each potentially undefined slot is already associated with a particular error thunk. Instead of wrapping the slot value up in the `Maybe` data type, the definedness check simply compares the slot value with the associated error thunk using pointer equality.

4 Adding Inheritance

It is possible to extend this translation further to allow a structure to inherit slots from other structures. For example, one might define variables which are just like definitions but

provide additional slots to store the type, signature, fixity, definition, etc of the variable. We extend the syntax slightly to specify which structures slots are being inherited from.

topdecl → `structure` $tycon_1, \ldots, tycon_n$ `=>` [`~`] *tycon* `where`
{ *structbody* [`;`] } [`deriving` `(` *tyclses* `)`] $(n \geq 1)$

For example, to define a type `Variable` which inherits slots from the type `Definition`, we write:

```
structure Definition => Variable where
  varType :: Signature
  varSignature :: Maybe Signature
  fixity :: Fixity
  definition :: Expression
```

The major change required to make this work is that the functions to select slots and update structures must be overloaded [9]. That is, instead of translating a structure definition into just a datatype and a collection of slot selection and update functions, structure definitions are translated into a type class with selection and update functions as methods, a new datatype and an instance of the datatype for that class. We use the same name for the type its corresponding class — this would normally be a syntax error since Haskell does not allow types and classes to share names.

For example, the definition of the structure `Definition` must be changed to define a type class (also called `Definition`) with methods

```
name, moduleName, unit :: Definition a => a -> String
...
(name=), (moduleName=), (unit=) :: Definition a => String -> a -> a
...
```

The old definition of the access functions is used to define an instance of the class `Definition` at the type `Definition`.

Similarly, the definition of the type `Variable` is used to define a type class `Variable`, and a data type `Variable` which is an instance of both `Definition` and `Variable`.

A structure may be either narrowed to a contained structure or widened to a containing structure. Widening is accomplished by adding undefined slots to the value. For a structure type `S`, the function `(-> S)` narrows a value from any type which includes `S` onto `S` and the function `(S ->)` widens a value of type `S` into any type containing `S`. The types of these operators are:

```
(-> S) :: S a => a -> S
(S ->) :: S a => S -> a
```

Translation:
```
instance S S' where
...
    (-> S) (MkS' x1 ... xn) = MkS n1 ... ni
    (S ->) (MkS x1 ... xm) = MkS' w1 ... wj
```
where n_i is the x in the corresponding slot and w_i is the corresponding x when the slot is part of S or $\perp$ otherwise.

For simplicity, widening does not invoke the defaulting mechanism to fill the new slots added by widening.

The most difficult change is in pattern matching. Since we do not know the exact type of the structure, the translation given in section 3.2 is no longer valid. The translation we implemented is:

Translation: The expression `case` e_0 `of (`$s1$ `=` p_1`, ...` sn `=` p_n`) -> `e`; _ ->` e' , for slot names $s1, \ldots sn$, is equivalent to:
```
let { x = e0; y = e' } in
case s1 y of { p1 -> ... case sn x of { pn -> e ; _ -> y } ...
               _ -> y }
```
where x, y, $x_1 \ldots x_k$ are new variables and x_s is the value of the slot named s.

This translation has the drawback that it may occasionally cause a space leak if any p_i is irrefutable. The problem is exactly that reported by Wadler [7]: slot extraction is only performed when the value of the slot is actually required; not when the pattern matching occurs. This can cause the entire structure to be retained when only one slot is required.

The following alternative translation would eliminate this space leak, but *may* make overloaded pattern matching more expensive. (This translation is for single inheritance. Extending it to handle multiple inheritance is straightforward but tedious.)

Alternative translation: If e_0 has type $S'\ \alpha \Rightarrow \alpha$, and S' has slots $s1, \ldots sn$, the expression `case` e_0 `of (`$s1$ `=` p_1`, ...` sn `=` p_n`) -> `e`; _ ->` e' , is equivalent to:
```
    let { x = e0; y = e' } in
    case (->S') x of { MkS' x1 ... xk ->
    case xs1 of { p1 -> ... case xsn of { pn -> e ; _ -> y } ...
                  _ -> y
    }}
```
where x, y, $x_1 \ldots x_k$ are new variables and x_s is the value of the slot named s.

4.1 Avoiding Inheritance

Inheritance is a powerful tool but its use presents two problems:

1. Since inheritance is implemented with the class system, using inheritance involves the same overhead that overloading functions entails. This overhead consists of both the instances needed to define an operation over a set of data types and the extra level of indirection needed to call overloaded functions. While the execution time overhead can be eliminated using type signatures to eliminate overloading, this is very burdensome for the programmer.

2. Inheritance also may prevent early detection of some errors. For example, given two structures

```
structure S1 where a1, b1  :: Int
structure S2 where a2, b2  :: Int

f (a1 = x, b2 = y) = x + y
```

   The definition of `f` is almost certainly incorrect since its argument must contain slots from two different structure types. However, this does not cause a type error since a third structure may later be declared (perhaps in a separately-compiled module) which includes both `S1` and `S2`.

   On the other hand, a type error does occur if we try to apply `f` to an argument of type `S1` (which is probably what the programmer intended to do.)

   If `S1` had not been overloaded, this error would have been caught when `f` was declared. (Providing the type signature `f :: S1 -> Int` would also have caught this error.)

We thus make inheritance optional: a structure declaration may indicate that the declared structure will not be inherited by any other structure. This is accomplished using a `~` in front of the structure name in the declaration:

```
structure S1 => ~S2 where s :: Int
```

The `~` prevents `S2` from being used as a class and allows any use of the slot `s` to precisely determine the typing of an update or pattern.

4.2 Multiple Inheritance and Defaulting

The Haskell type class system allows a class to have multiple superclasses. Since structures are translated into type classes, our translation naturally allows *multiple inheritance:* a structure is allowed to inherit slots from any set of other structures.

In the type class system, defaults can only apply to methods directly associated with a class, not those inherited from superclasses. This avoids ambiguity over which default to apply when the same method is inherited via several routes (e.g. the standard class `Integral` inherits `Ord` via both the `Ix` class and the `Real` class).

We have chosen to relax this rule for structures. Structure declarations may define default methods for inherited slots. The following rule is used to avoid ambiguity:

If a structure inherits a slot `s`, it may either define a new default for `s` or use the default associated with the first structure in the list of included structures containing `s`.

4.3 The Polymorphic Inheritance Problem

The reader may have noticed that the syntax for structure declarations does not allow both polymorphism and inheritance. This is to avoid the following limitation of Haskell's type system.

The declarations generated by:

```
structure S1 a where
 s1 :: a

structure S2 b where
 s2 :: b

structure S1 a, S2 b => S3 a b
```

would be:

```
data S1 a = MkS1 a
data S2 b = MkS2 b
data S3 a b = MkS3 a b

class S1 s where
  s1 :: s a -> a

class S2 s where
  s2 :: s b -> b

-- instances for S1, S2 omitted
instance S1 (S3 b) where
  s1 (MkS3 x _) = x

instance S2 (S3 a) where
  s2 (MkS3 _ x) = x
```

This "class declaration" is not legal Haskell since the type variable `s` must be instantiated with a type constructor rather than a type. This may appear to be legal using constructor classes, but the instance declaration for `S1` will still not work.

For now we simply prohibit the inheritance of polymorphic structures but allow polymorphic structures and unrestricted inheritance of non-polymorphic structures. It remains to be seen whether this is excessively restricting in real programs.

5 Alternatives and Related Work

Our system is an experiment, not a finished product. Having the experience of carrying an implementation all the way through and using it on a number of real applications, including the Yale debugger and a prototype GUI system, we can assess our design and consider alternatives.

5.1 The Namespace Issue

Our placement of slot names into the value namespace is a significant difference from languages such as C, Pascal, or ML. Using a separate namespace for each structure in the manner of C is not possible, however, because this depends on a bottom-up style of type inference which determines which type of structure is involved before resolving field names.

In practice, we have found that placing selector functions in the value namespace makes it almost essential to use long field names. For example, the structure `Point` defined by

```
structure ~Point where x, y :: Int
```

introduces two top-level function names `x` and `y` which the programmer is likely to want to use for other purposes. We adopted the convention of using the structure name as a prefix for the field name. For example, we would normally choose slot names `pointX` and `pointY` instead of `x` and `y`.

This problem could be reduced by providing special syntax for selector functions — avoiding the need to place selector functions in the value namespace— but this would not completely avoid the problem: all slot names would still be in the same namespace.

A more radical solution is used in ML which allows "labels" to be shared among different records. These labels do not carry typings in the same way the slot names do. Instead, they simply attach names to tuple components. Implementing records using shared labels would require significant changes to the syntax and further complicate the type system.

5.2 Default values

In our experience, some sort of defaulting mechanism is essential. This allows new fields to be inserted into a structure without changing all references to the associated constructor. Although not often used, the expressiveness of mutually recursive slot initialization can be very useful and seems to be more in the Haskell spirit than restricting defaults values to constants or imposing some sort of evaluation order on the default computation.

5.3 Uninitialized Slots

Though easy to implement, the ability to detect uninitialized structure slots is somewhat dubious. To date, our only use of this feature has been to allow the derived `Text` instances for structures to skip over uninitialized slots.

The need to detect uninitialized slots could be eliminated by making it impossible to leave a slot uninitialized. This could be done by changing the syntax of structure creation to require a list of slot names and values (rather than allowing any expression of the right type). It would then be possible for the compiler to check that every slot had either a default value or an explicitly provided value. A similar restriction is imposed in ML [3], where it is required by the combination of strict semantics and type safety.

5.4 Pattern Matching

Pattern matching in Haskell lacks the extensibility of other language features. It would certainly be better to add a general purpose mechanism flexible enough to define structure pattern matching than to add structure pattern matching as a special case, as in our implementation. Sadly, Wadler's "views" [8, 1] would not be flexible enough to handle this case.

In practice, we found that we didn't use pattern matching very much, preferring to use selector functions to extract slots at the place where they are needed rather than at the head of a function. This may be caused by a number of factors: our familiarity with this style of programming from other languages that support records; our use of long field names (section 5.1); the fact that structure pattern matching is generally not connected with control flow; or our use of structures in big, complicated programs that solve real problems instead of in highly polished classroom examples.

5.5 Allowing Polymorphic Inheritance

There appears to be a simple extension to constructor classes which would allow polymorphic inheritance. The problem with constructor classes is that only those types which are curried applications of a type constructor are available. Thus, for a type `T a b`, constructor classes can make use of `T`, `T a`, and `T a b` as types. Expanding the implicit currying, these types are `\a b -> T a b`, `\b -> T a b`, and `T a b`. Unfortunately, polymorphic inheritance requires a type such as `\a -> T a b`. We conjecture that adding a limited lambda to the type language is possible: this lambda is needed only to permute the arguments to the type constructors.

5.6 Syntax Issues

Using similar syntax for update functions (which are functions) and structure patterns (which match data values) is somewhat irregular. In hindsight, it would be possible to drop the parenthesis in single update functions and to drop multiple update functions. Where one currently writes update functions such as `(moduleName = m, name = nm)`, one would instead write `(moduleName = m . name = nm)`.

Our use of special syntax such as `(s=)`, `(=s)`, `(-> S)` and `(S ->)` is somewhat contorted. An alternative would be to indulge in name mangling (deriving one name from another) as in Common Lisp. (For example, the function `setFoo` would be used to alter the values of slot `foo`.) However, no other Haskell feature uses name mangling so we hesitate to add this.

5.7 Record Types

An entirely different system can be constructed using labeled records and subtype inference [5, 4]. The advantage of such as system would be that structure declarations would be unnecessary. While type systems have been proposed featuring subtyping based on extensible records, these have two disadvantages: these require a fundamental change to the Haskell type system and it may be difficult to generate efficient record operations using these systems.

5.8 Generalizing to Arbitrary Datatypes

The structures considered in this proposal are just syntactic sugar for tuples; but, Haskell's datatypes allow one to define a "sum of tuples". It would be straightforward to adapt the inheritance-free translation in section 3 to allow one to define field names for arbitrary datatypes. For example, given the datatype:

```
data Expr = Lambda (arg :: Var) (body :: Expr)
          | App (fun :: Expr) (arg :: Expr)
          | Var (v :: Var)
```

one could use pattern matching such as:

```
eval env (Lambda (arg = v, body = e)) = \x. eval ((v,x):env) e
eval env (App (fun = f, arg = a))     = (eval env f) (eval env a)
eval env (Var (v = x))                = lookup env x
```

5.9 Object-Oriented Programming

The ability to inherit structure slots is a step toward a more object-oriented programming paradigm. However, when we used our structure system in a GUI system in an object-oriented style, a number of deficiencies became obvious.

First, the classes defined for structures contain only slot accessing functions. To add other class methods (as with C++ virtual functions), we were forced to add an extra class for each structure type. That is, for a structure S (which defines a class S), we added the class `S => S'` to hold methods associated with structures inheriting from S. This was very unsatisfactory — it would be much nicer to be extend structure definitions to directly include these methods.

Dynamic binding, which would allow methods (dictionaries) to be attached directly to data values, is not available in Haskell without some sort of existential typing. This makes non-homogeneous lists impossible in standard Haskell.

The coercion functions were very useful — these allow objects to be moved up or down the class hierarchy so as to dispatch methods associated with other types.

A more general object-oriented extension to Haskell would eliminate the need for slot inheritance at the structure level. Provided any extra overhead could be eliminated by the compiler, such an extension may be preferable to using the inheritance mechanism described here.

5.10 Code Generation

We have found that three factors significantly affect the quality of the generated code:

1. Inlining selection and update functions eliminates a function call and allows further optimizations to be performed. Inlining the initialization function avoids constructing and destructuring many partial records.

2. Using pattern matching on function arguments produces code that is both more efficient and less likely to leak space than if selector functions are used. The reason is simple: pattern matching is performed when the function is called whereas selection functions are only executed when the selected value is evaluated. Exactly the same difference occurs if programmers use pattern matching on lists instead of `head` and `tail`.

3. Avoiding overloading (whether by shunning inheritance or by providing explicit type signatures) eliminates dictionary lookups and allows selection and update functions to be inlined.

   Restricting ourselves to single inheritance would allow a more efficient implementation of inheritance: inherited slots could be placed at the same offset from the start of a structure as in their parents allowing exactly the same code sequence to be used for selecting a slot — no matter what it's type. This optimisation would eliminate the need to pass dictionaries around; greatly improving performance.

By choosing the best options (inline structure operations, use pattern matching and avoid overloading), we are able to generate exactly the same code as if no abstraction mechanisms had been used.

6 Conclusions

Our experience of being able to name fields has been entirely positive — we feel that it significantly improves the readability and maintainability of our programs. Having an elegant notation for updates is also very useful. Programs using these features are easier to maintain and the code is very readable.

The best way to deal with inheritance is not yet known. A more advanced object-oriented extension to Haskell may provide the same capabilities we have implemented. Simplifying to a single inheritance style would eliminate the performance problems introduced through the use of the class system.

Much of the implementation baggage could be eliminated by removing non-constant defaults and inheritance. This would make structure creation trivial: an update is applied to a structure containing the constant defaults. No class or instance declarations would be generated by structures; only data declarations. No support functions would be required — all structure operations could be expanded inline. Such a stripped-down system would address many, but not all, of the engineering issues described earlier. At a minimum, such a system should be considered for Haskell 1.3.

Acknowledgments

We are grateful to Warren Burton, Mark Jones, and Randy Hudson for their comments on an early design of this system. Sandra Loosemore and others in the Yale Haskell group also provided valuable assistance.

References

[1] FW Burton and RD Cameron. Pattern matching with abstract data types. *Journal of Functional Programming*, 3(2):171–190, April 1993.

[2] P Hudak, SL Peyton Jones, PL Wadler, Arvind, B Boutel, J Fairbairn, J Fasel, M Guzman, K Hammond, J Hughes, T Johnsson, R Kieburtz, RS Nikhil, W Partain, and J Peterson. Report on the functional programming language Haskell, Version 1.2. *ACM SIGPLAN Notices*, 27, May 1992.

[3] R Milner, M Tofte, and R Harper. *The definition of Standard ML.* MIT Press, 1990.

[4] Atsushi Ohori. A compilation method for ML-style polymorphic record calculi. In *Principles of Programming Languages*, pages 154–165. ACM, January 1992.

[5] D Rémy. Typechecking records in a natural extension of ML. In *Principles of Programming Languages*, pages 242–249. ACM, January 1989.

[6] GL Steele. *Common Lisp — The Language.* Digital Press, 2nd edition, 1994.

[7] PL Wadler. Fixing a space leak with a garbage collector. *Software — Practice and Experience*, 17(9):595–608, 1987.

[8] PL Wadler. Views — a way for pattern matching to cohabit with data abstraction. Technical Report 34, Programming Methodology Group, Chalmers University, Sweden, March 1987.

[9] PL Wadler and S Blott. How to make ad-hoc polymorphism less ad hoc. In *Principles of Programming Languages.* ACM, January 1989.

Haskell++: An Object-Oriented Extension of Haskell

John Hughes and Jan Sparud,
Department of Computer Science,
Chalmers University,
Göteborg, SWEDEN.
{rjmh,sparud}@cs.chalmers.se

June 2, 1995

1 Introduction

Lazy functional languages such as Haskell[Hud92] provide excellent support for writing re-useable code. Polymorphism, higher-order functions, and lazy evaluation are all key contributing features:

- instead of writing many *sort* functions at different types, we re-use one;
- instead of writing many functions which recurse over lists, we capture common recursion patterns as higher-order functions *map*, *foldr* and so on, and just re-use them;
- instead of writing many loops that iterate n times, we write loops producing infinite lists and re-use *take* n to count the iterations.

This re-useability is reflected, for example, in the very heavy use that Haskell programs make of functions from the standard prelude. Indeed, we have argued elsewhere[Hug89] that these features are largely responsible for the improved productivity that functional programming offers.

Haskell's overloading system [WB89] also contributes to code re-useability. For example, most numeric functions in Haskell programs can be re-used with *any* implementation of numbers. Although in this case overloading can be regarded as syntactic sugar for parameterising numeric functions on the implementations of the arithmetic operations, the sugar is important because it makes re-useable code easy to write. Most programmers would probably regard passing the operations explicitly as unacceptably clumsy, and therefore wouldn't do it. The result: less re-useable code.

However, there is a form of re-use which Haskell does not support. Suppose we define a type T which is an instance of class C, and now we want to define a new type T' which contains a T and some additional components. Suppose further we want to make T' an instance of the same class. It is quite likely that the new components will only be significant for *some* of the class operations, and we would therefore like to write an instance declaration for T' where we only define these operations explicitly, and re-use T's definitions for the others. Sadly this is impossible: instance declarations in Haskell must define *all* of the methods in the class[1]. T' cannot *inherit* method definitions from T.

This kind of re-use is of course supported by *object-oriented* languages. Although the object-oriented languages used in practice are imperative, there is no shortage of functional variants in the theoretical literature — see for example [jfp94]. But these variants require relatively powerful and complicated type systems. For example, Pierce and Turner [PT94] show how an object-oriented language can be simply translated into $F^{\omega}_{\leq}$, that is λ-calculus with higher-order polymorphism and subtyping.

Such a translation could be used directly to implement a functional object-oriented language, given a compiler for the target language, but unfortunately good compilers for $F^{\omega}_{\leq}$ are in short supply. In this paper we show how to translate an object-oriented language into Haskell instead. Our translation is rather similar to Pierce and Turner's, but where they use subtyping to allow an inherited method to be applied at different types, we use Haskell's overloading with automatic generation of suitable instances. We have implemented a simple pre-processor which translates an object-oriented extension of Haskell, 'Haskell++', into vanilla Haskell which can then be compiled with any of the existing compilers. We believe that, like Haskell's class system, our syntactic sugar can significantly improve code reuseability in practice.

2 An Introduction to Haskell++

We begin by giving an informal presentation of Haskell++, and showing how some well-known object-oriented examples can be programmed.

2.1 Object Classes and Instances

Haskell++ extends Haskell by providing a new kind of *class*, whose methods may be inherited from one instance to another. These classes are defined by an **object class** declaration. For example, let us declare an object class of points, with methods for extracting the coordinates, moving the point, and showing the point.

[1] Haskell's default methods are not useful here: we want to re-use T's methods, not some general class-wide method.

```
object class Point where
  x  :: Int
  y  :: Int
  mv :: Int → Int → self
  sh :: String
```

All the methods of an object class must be applied to an object of an appropriate type, and the type of this object is always called *self*. Since this first parameter must always be present, it is not given explicitly in the object class definition. The type of the *mv* method declared above, for example, is actually

$$mv :: Point\ self \Rightarrow self \to Int \to Int \to self$$

Moreover, since the type of the object operated on is always called *self*, there is no need for a type variable in the **object class** line.

It may seem a little odd to treat the first parameter of each method specially. One unfortunate consequence is that we cannot define overloaded methods to *create* objects in the first place. We are forced to define a different object creation function for each object type, and these functions *cannot* be inherited. But on the other hand, it is a little unclear what inheriting such a method would mean, and the restriction is shared with other object oriented languages such as Eiffel [Mey88]. An advantage is that we can never suffer from ambiguous overloading. Our implementation of inheritance depends on the first parameter having type *self*, as we will explain below. To permit the user to write '*self* →' in each method type declaration, and then reject anything else as an error, would be perverse.

To get any further we need a type which can be made an instance of this class. Type definitions are as in Haskell, for example

```
data VanillaPoint = P Int Int
```

Now we can make *VanillaPoint* an instance of class *Point* by making an **object instance** declaration. Just as the first parameter of each method was implicit in the object class definition, so it is implicit in the object instance definition. But in order to refer to the components of the object, we give a pattern that it must match in the declaration header. Within the method definitions we can refer to the variables bound in this pattern.

```
object instance Point (P x0 y0 :: VanillaPoint) where
  x = x0
  y = y0
  mv x' y' = self (P x' y')
  sh = show (x0, y0)
```

The occurrence of *self* in the *mv* method requires explanation. Remember that these methods may be inherited by other instance declarations. That means that the *mv* method defined here may very well be applied to types other than

VanillaPoint! In other words, even though we are defining the *VanillaPoint* instance here, the type we called *self* above need not be *VanillaPoint*. To define *mv* by

$$mv\ x'\ y' = P\ x'\ y'$$

would therefore be a type error: the result is of type *VanillaPoint* but should be of type *self*. In order to construct values of the right type, Haskell++ provides a *function*, also called *self*, which can be used only in method definitions and whose type in this case is

$$self :: VanillaPoint \rightarrow self$$

Every type which inherits from *VanillaPoint* will contain a *VanillaPoint* component. The *self* function builds a copy of the object that the method it is used in is applied to, in which the *VanillaPoint* component is replaced by its argument. It provides a mechanism for Haskell++ methods to to invoke other methods in the same object, and to construct modified versions of the object they are applied to.

2.2 Inheritance

Now suppose we want to define a type of points which also carry a colour. We define

```
data Colour = Red | Yellow | Blue deriving Text
data ColourPoint = CP Colour VanillaPoint
```

Let us make *ColourPoint* an instance of class *Point*. The only method affected by the colour is *sh*: we want to show the colour too. We would like just to inherit the other methods from *VanillaPoint*. To do so, we define

```
object instance Point (CP c p :: ColourPoint)
    inheriting x, y, mv from p where
    sh = super_sh ++ ", " ++ show c
```

Now when *mv*, for example, is applied to a *ColourPoint* it will just move the *VanillaPoint* component, leaving the *Colour* unchanged. For example,

$$mv\ 1\ 2\ (CP\ Red\ (P\ 0\ 0)) = CP\ Red\ (P\ 1\ 2)$$

The *mv* method inherited from *VanillaPoint* rebuilds a *ColourPoint* by applying the *self* function, which in this case is (*CP Red*).

Even when defining methods explicitly, we can refer to the inherited methods under names beginning with *super_*. For example, here we have defined the *sh* method for *ColourPoints* in terms of the inherited *sh* method for *VanillaPoints*:

$$sh\ (CP\ Red\ (P\ 1\ 2)) = \text{"(1,2), Red"}$$

2.3 Virtual Methods

Inheritance behaves rather subtly when one object method is defined in terms of another. As an example, suppose we define a new class *UpperPoint*, whose objects can be shown in upper case. We shall provide two alternative methods for doing so, so we declare

$$\begin{array}{l}\textbf{object class } \mathit{Point} \Rightarrow \mathit{UpperPoint} \textbf{ where} \\ \quad \mathit{shU} \;::\; \mathit{String} \\ \quad \mathit{shU'} \;::\; \mathit{String}\end{array}$$

Here we require that instances of *UpperPoint* are also instances of *Point*.

Now let us make *VanillaPoint* an instance of this class:

$$\begin{array}{l}\textbf{object instance } \mathit{UpperPoint}\ (p :: \mathit{VanillaPoint}) \textbf{ where} \\ \quad \mathit{shU} = \mathit{map}\ \mathit{toUpper}\ (\mathit{sh}\ (\mathit{self}\ p)) \\ \quad \mathit{shU'} = \mathit{map}\ \mathit{toUpper}\ (\mathit{sh}\ p)\end{array}$$

Of course, these methods are defined in terms of *sh*.

We can inherit these definitions for *ColourPoints*:

$$\begin{array}{l}\textbf{object instance } \mathit{UpperPoint}\ (\mathit{CP}\ c\ p :: \mathit{ColourPoint}) \\ \quad \textbf{inheriting } \mathit{shU}, \mathit{shU'} \textbf{ from } p\end{array}$$

But *ColourPoints* have a different *sh* method from *Points*! The question is, do *shU* and *shU'* 'see' the *ColourPoint sh* method, even though they are inherited from *VanillaPoint*? In object-oriented languages the answer is 'yes', and indeed, classes often contain so-called *virtual methods*, which are left undefined at the 'vanilla' instance, and are intended to be overridden at *every* inheriting type.

In Haskell++ the behaviour of *shU* and *shU'* is different. When *shU* is applied to a *ColourPoint*, it uses the *self* function to reconstruct a *ColourPoint* and applies *sh* to that. It therefore 'sees' the new *sh* method:

$$\mathit{shU}\ (\mathit{CP}\ \mathit{Red}\ (P\ 1\ 2)) = \text{"(1,2), RED"}$$

On the other hand, *shU'* applies *sh* directly to a *VanillaPoint*. It does not see the new method therefore:

$$\mathit{shU'}\ (\mathit{CP}\ \mathit{Red}\ (P\ 1\ 2)) = \text{"(1,2)"}$$

This difference in behaviour helps explain why we need *super_*. Looking back at the definition of *sh* for *ColourPoints*, the reader may have wondered why we wrote

$$\mathit{sh} = \mathit{super\_sh} +\!\!+ \text{", "} +\!\!+ \mathit{show}\ c$$

instead of

$$\mathit{sh} = \mathit{sh}\ p +\!\!+ \text{", "} +\!\!+ \mathit{show}\ c$$

Why do we need special syntax to call an inherited method, instead of just applying the method to the component we inherit from?

In this case the two definitions are actually equivalent, because the *sh* method for *VanillaPoints* does not use any other object methods. But suppose *sh* were defined in terms of the x and y methods, instead of the x_0 and y_0 components. These methods might be overriden at other types — for example, we might define a type *XAxisPoint*, inheriting from *ColourPoint*, whose y method always returns 0. Does the *sh* method for *XAxisPoints*, inherited from *ColourPoints*, see the new y method or not? With the definition in terms of *super_sh*, the answer is 'yes': the *sh* method inherited from *VanillaPoints* sees the modified y method. But with the definition in terms of *sh p* the answer is 'no': here we simply apply *sh* to a *VanillaPoint*, and all other information is lost.

2.4 Multiple Inheritance

It is straightforward for a type in Haskell++ to be an instance of several object classes, and to inherit from more than one component. For example, let us define a class of coloured objects:

```
object class Coloured where
  colour    :: Colour
  setColour :: Colour → self
```

The simplest instance is just the type *Colour*:

```
object instance Coloured (c :: Colour) where
  colour = c
  setColour c' = self c'
```

Now it is easy to make *ColourPoints* into *Coloured* objects, by inheriting from the *Colour* component:

```
object instance Coloured (CP c p :: ColourPoint)
  inheriting colour, setColour from c
```

ColourPoints now inherit in two different ways.

This is actually only a limited form of multiple inheritance, because we inherit the operations of each class quite independently, in separate instance declarations. We can't, for example, define a *sh* method for *ColourPoints* which uses both the inherited *sh* method from *VanillaPoint* and the inherited *colour* method from *Colour*.

2.5 Dynamic Binding

A very useful aspect of object-oriented languages is so-called *dynamic binding* of methods, which allows the overloading implicit in a method application to

be resolved at run-time. For example, one may make a list containing both *VanillaPoint*s and *ColourPoint*s, and indeed any other instance of the *Point* class, and then map the *sh* method over the list. Thanks to dynamic binding, the appropriate *sh* method is invoked for each element.

In Haskell, and also Haskell++, one cannot even build such a list because the elements have different types. Pierce and Turner overcome this problem by concealing the representation type of *Point* objects using an existential type, so that every kind of point actually has the same type. Läufer and Odersky have shown how existential types can be added smoothly to ML [Läu92, LO92] and Haskell [Läu94], and how the resulting extension supports dynamic binding. Existential types are not yet a part of standard Haskell, but Augustsson has implemented them in `hbc`.

With Läufer's extension, we can define a type

$$\textbf{data}\ \mathit{DynamicPoint} = \mathit{Point}\ p \Rightarrow \mathit{DP}\ p$$

The *DP* constructor can be applied to *any* type p in class *Point*, but *the type of the result does not depend on* p! So if *vp* is a *VanillaPoint*, and *cp* is a *ColourPoint*, then we can form both *DP vp* and *DP cp*, and in both cases the result is of type *DynamicPoint*. We can therefore place both values in the same list. Of course, when we use a *DynamicPoint*, all we know about the component is that it is in class *Point*, and so the only operations we may apply to it are the corresponding class operations. When we do so, the appropriate instance is selected dynamically, implementing dynamic binding.

This extension is quite independent of Haskell++, but if we are using the preprocessor with a compiler supporting existential types, then we can conveniently make *DynamicPoint* an instance of class *Point* using inheritance:

$$\begin{aligned}&\textbf{object instance}\ \mathit{Point}\ (\mathit{DP}\ p :: \mathit{DynamicPoint})\\ &\quad \textbf{inheriting}\ x, y, \mathit{mv}, \mathit{sh}\ \textbf{from}\ p\end{aligned}$$

3 Translating Haskell++ to Haskell

The fundamental problem in translating Haskell++ to Haskell is the implementation of inheritance. Our approach is as follows: any object which inherits methods from, say, *VanillaPoint*, must first be decomposed into a component of type *VanillaPoint*, and a function which rebuilds the rest of the object from that component (that is, *self*). So method bodies in Haskell++ actually have *two* hidden parameters: the *self* function and the component inherited from. Object instance declarations are translated into Haskell instance declarations just by adding these two parameters to every method. The 'object pattern' given in an object instance declaration is of course used to match the object parameter.

For example, the object instance declaration for *VanillaPoint*s

```
object instance Point (P x0 y0 :: VanillaPoint) where
  x = x0
  y = y0
  mv x' y' = self (P x' y')
  sh = show (x0, y0)
```

is translated into the Haskell instance declaration

```
instance Point VanillaPoint where
  xBody self (P x0 y0) = x0
  yBody self (P x0 y0) = y0
  mvBody self (P x0 y0) x' y' = self (P x' y')
  shBody self (P x0 y0) = show (x0, y0)
```

Notice that the operations in the generated class are actually 'method bodies', not the methods themselves.

Correspondingly, object class declarations are translated by renaming the methods and adding the two hidden parameters to the type of each method. For example,

```
object class Point where
  x  :: Int
  y  :: Int
  mv :: Int → Int → self
  sh :: String
```

is translated into

```
class Point obj where
  xBody :: Point self ⇒ (obj → self) → obj → Int
  yBody :: Point self ⇒ (obj → self) → obj → Int
  mvBody :: Point self ⇒ (obj → self) → obj → Int → Int → self
  shBody :: Point self ⇒ (obj → self) → obj → String
```

Notice that the type parameter of the class is *not* the type *self*, it is a new type variable *obj* representing the type of the 'object pattern' in an instance declaration. When we define a particular instance, it is of course this type that we fix — but every instance definition must still be polymorphic in the type *self*. A little care is needed here to translate type contexts correctly, both in the types of individual methods and in the class header, but we leave the details to the reader.

The object methods themselves are defined by using the trivial decomposition of an object into itself and the identity function:

```
x obj = xBody id obj
y obj = yBody id obj
mv obj = mvBody id obj
sh obj = shBody id obj
```

These definitions are generated when the object class declaration is processed.

Inheriting a method is equivalent to defining it to be equal to the corresponding *super_* method, so for example

```
object instance Point (CP c p :: ColourPoint)
  inheriting x, y, mv from p where
  sh = super_sh ++ ", " ++ show c
```

is equivalent to

```
object instance Point (CP c p :: ColourPoint)
  inheriting from p where
  x = super_x
  y = super_y
  mv = super_mv
  sh = super_sh ++ ", " ++ show c
```

So it only remains to explain the translation of *super_* methods. They can only be used within the scope of *self* and the object pattern. We can therefore translate an occurrence of *super_mv*, for example, into an application of *mvBody* to the object component we're inheriting from, and a suitably extended *self* function. In this example, *self* maps *ColourPoints* to the *self* type, and we must pass the *VanillaPoint mv* method a function from *VanillaPoints* to this type. We can construct it as $self \circ (\lambda p \rightarrow (CP\ c\ p))$, and in general the appropriate *self* function is constructed similarly by composing the outer *self* with a λ-expression constructed from the object pattern and the name of the component we are inheriting from. So the generated definition for the *ColourPoint mv* method is

$$mvBody\ self\ (CP\ c\ p) = mvBody\ (self \circ (\lambda p \rightarrow (CP\ c\ p)))\ p$$

and similarly for the other methods[2].

4 A Larger Example

To test the object-oriented features gained by combining the Haskell class system and existential types [Läu94] we had already written a simple drawing program in an object-oriented style. We have since rewritten the program in Haskell++.

In the Haskell version we defined a class for graphical objects with methods to draw them, move them, get information about them etc.

[2]Note that *self* rebuilds the structure of the object above the component that has changed; as always in a functional language, modifying a part of a structure incurs a certain cost. Analyses to detect single-threading could potentially be used here to transform *self* into a destructive update.

```
class Object a where
    origin      :: a → Point
    newOrigin   :: a → Point → a
    move        :: a → Point → a
    draw        :: a → [DrawCommand]
    ...
```

To introduce a new graphical object type, we define a Haskell datatype for it and then make the type an instance of the graphical object class. One benefit of this approach is that when extending the program with a new object type, all changes are made in one place. This is really a consequence of the object oriented methodology used. On the other hand, if we want to extend the class with a new method, then every instance declaration must be modified — and they may be spread over several different files.

In our simple program we defined object types for points, lines, rectangles, and circles. Here are the instance declarations for lines and rectangles.

```
data Line = Line Point Point
instance Object Line where
    origin (Line p1 _)          = p1
    newOrigin (Line _ p2) p     = Line p p2
    move (Line p1 p2) d         = Line (move p1 d) (move p2 d)
    draw l                      = [DrawLine l]
    ...
data Rect = Rect Line
instance Object Rect where
    origin (Rect l)             = origin l
    newOrigin (Rect l) p        = Rect (newOrigin l p)
    move (Rect l) d             = Rect (move l d)
    draw l                      = [DrawRectangle l]
    ...
```

(Rectangles are represented by their diagonal).

From these declarations we can see a drawback of the approach: it is clumsy to re-use properties of one instance of the graphical object types in another. For example, rectangles and lines have a common property in that they are characterised by two corner points. If we define a *move* function for lines, we would like to be able to use that function also when moving rectangles. To do this we must explicitly define a function in the rectangle instance that uses the *move* function from the line instance.

In order to manipulate objects of different graphical types together, we also defined an existential type and made it an instance of the graphical object class (see 2.5). The instance declaration is straightforward but boring to write; every class method is defined in terms of the corresponding method for the contained 'object'.

```
data O = (Object obj) ⇒ O obj
instance Object O where
    origin (O obj)          =  origin obj
    newOrigin (O obj) p     =  O (newOrigin obj p)
    move (O obj) d          =  O (move obj d)
    draw (O obj)            =  draw obj
    ...
```

Values of the different graphical object types were then embedded into this existential type so that they, e.g., could be put into a list. For example,

```
p1 = Point 0 0
p2 = Point 100 200
p3 = Point 50 100

objects = [O p3, O (Line p1 p2), O (Rect (Line p2 p3))]
```

One can now map a function over all objects in the list. Of course, the only interesting functions to use here are the class methods, since other functions cannot do anything with the values in the existential type. For example,

```
newObjects = [move obj (Point 30 50) | obj ← objects]
drawing = concat (map draw newObjects)
```

Thanks to dynamic binding, the *move* and *draw* methods used depend on the actual type of the object contained in each existential value.

After rewriting the program using Haskell++, the definition of the graphical object class looks like:

```
object class Object where
    origin      :: Point
    newOrigin   :: Point → self
    move        :: Point → self
    draw        :: [DrawCommand]
    ...
```

We can now exploit inheritance to reuse method definitions in several instances. In our case the rectangle instance can inherit the *origin*, *newOrigin*, and the *move* functions from its line component (and lines can inherit from their first point).

```
data Line = Line Point Point
object instance Object (Line p1 p2 :: Line)
   inheriting origin, newOrigin from p1 where
    move d  =  self (Line (move p1 d) (move p2 d))
    draw    =  [DrawLine (Line p1 p2)]
    ...
data Rect = Rect Line
object instance Object (Rect l :: Rect)
   inheriting origin, newOrigin, move from l where
   draw = [DrawRectangle l]
   ...
```

We derive a number of benefits from using Haskell++. When we define a new object type that is very similar to an existing object type, it is easy to inherit most methods and redefine the few ones that need to be changed, saving programming effort. Moreover, if the type of a class method is changed, then the method definitions have to be changed in every instance when using plain Haskell. But in Haskell++ no changes are necessary in those instances which inherited the method — only in instances where it is really redefined. Since less needs to be written, the risk of errors is also reduced.

We exploited Haskell++'s virtual methods to define a *create* operation, that uses mouse dragging to place a new shape on the canvas. The *create* method is defined in the *Line* instance and inherited by every descendant, but makes use of the *draw* method that is redefined in each instance.

The Haskell++ version uses an existential graphical object type in just the same way as the Haskell one, but the corresponding instance declaration becomes very simple: it just inherits everything!

```
data O = (Object obj) ⇒ O obj
object instance Object (O obj :: O)
   inheriting origin, newOrigin, move, draw, ... from obj
```

In this example in particular, it is somewhat inconvenient to have to list all the methods one wants to inherit in an object instance. It would be better if all methods that are not explicitly redefined in an object instance declaration could be inherited automatically.

The graphical objects module in our example program defines one object class with 13 methods. There are five instances, in which 23 methods are defined explicitly, 32 are inherited (and ten are virtual). The number of source lines decreased from 160 to 90 when moving from Haskell to Haskell++ — a reduction of over 40%.

Of course, these figures very much depend on the application. An important point is that the lines that we do not need to write anymore are very 'mechanical': they are boring to write and prone to errors, and labour intensive to change should that be necessary.

5 Related Work

5.1 Haskell Classes

Haskell classes already provide *overloading*, which is one aspect of object orientation. But Haskell classes do not support any form of *inheritance*: the method definitions in one instance cannot be inherited by another. Haskell++ object classes differ in providing inheritance. On the other hand, method types are restricted to the form $self \to \tau$, and within method definitions *self* is an ***abstract*** type. That is, the only way to manipulate values of type *self* (except the first parameter) is with the methods of the class, ***even within method definitions.*** As a result object classes only support so called 'weak binary methods', a restriction which does not apply to Haskell classes.

5.2 Läufer and Odersky's Existential Types

We have already described Läufer and Odersky's addition of existential types to Haskell briefly above (section 2.5). Läufer has discussed its application to object-oriented programming, including the points and colour points example, in [Läu94]. By 'wrapping' objects of different types, such as points and colour points, in a value of existential type, Läufer can for example form heterogenous lists, and apply methods to the list elements with the right implementation being selected dynamically. We borrow this technique in section 2.5.

However, since they do not extend Haskell's class mechanism, Läufer and Odersky must still define every instance method explicitly: there is no automatic inheritance. Furthermore, they do not provide virtual methods: there is no way for a method defined on points to use a method defined on colour points.

5.3 Pierce and Turner's Translation

Pierce and Turner describe a translation from a simple object-oriented language into system $F^{\omega}_{\leq}$, with an implementation of inheritance strongly reminiscent of ours [PT94]. But whereas we inherit from *obj* to *self* by passing the inherited method bodies an *obj* and a function $obj \to self$, Pierce and Turner pass it three parameters with types

- *self*,
- $self \to obj$,
- $self \to obj \to self$

namely the original object, a function to extract the component we are inheriting from, and a function to replace this component. In effect, we pass the applications of these two functions to the original object, rather than passing the object and functions separately. Pierce and Turner's idea is clearly more

general, but we are unable to adopt it because in our framework *no such functions need exist*!

The problem is caused by existential types. Recall the type of dynamic points

$$\text{data } \mathit{DynamicPoint} = \mathit{Point}\ p \Rightarrow \mathit{DP}\ p$$

which inherits all its methods from p. Our implementation of inheritance defines *DynamicPoint* methods by pattern matching on $(\mathit{DP}\ p)$ and invoking the corresponding method on p. The result is always of a type which does not involve p, and so the definition is well-typed. But using Pierce and Turner's idea, we would need to construct a *function* with type $\mathit{DynamicPoint} \to p$, which cannot be done because the existentially quantified type p escapes from its scope.

Pierce and Turner represent *all* object types as existential types, whereas we leave it to the Haskell++ programmer to choose whether to use existential types or not. This probably explains our difficulty in the previous paragraph. Pierce and Turner's goal is to *translate* an object-oriented language into a functional one, while ours is to *extend* an existing language. To put it another way, we are very concerned that our translation should be the identity in almost all cases! Pierce and Turner need have no such concern. In particular, we have chosen to give types exactly the same meaning in Haskell++ as in Haskell, and therefore eschewed hidden existentials.

A minor difference between their translation and ours is that they use subtyping to allow inherited methods to be applied at many different types, while we use Haskell's overloading mechanism. As a consequence we are obliged to generate suitable instance declarations for inherited methods automatically.

6 Conclusions

Haskell++ extends Haskell with object classes, whose instances may inherit methods from one another. It is a rather minimal extension of Haskell: there are no new 'object types' or 'object expressions', for example. The only new features are object class and instance declarations.

Consequently the Haskell++ programmer must define object types using the existing Haskell type definition mechanism. Some may regard this as 'hair shirt' object-oriented programming. We prefer to say the new features are well integrated with the old — any Haskell type may be declared to be an instance of an object class.

The translation of Haskell++ to Haskell is straightforward. But the translations are sufficiently clumsy that few programmers would write them by hand. Therefore we believe that our 'syntactic sugar' leads to a real improvement in code reusability in practice.

We have tested Haskell++ in a non-trivial example, a simple object-oriented drawing program. The results show a significant improvement in code reuse

compared to using existential types alone.

Our translator keeps no context information and therefore needs all inherited methods to be named explicitly, which is a little unusual for an object-oriented language. This can become tedious, especially when a new method is added to an object class and all instance declarations have to be changed. An easy extension would automatically inherit all methods which are not explicitly defined.

Initial experience of Haskell++ is encouraging, but much more work is required to establish whether a combination of object-oriented and functional programming is truly valuable in practice.

A prototype translator is available from the authors.

References

[Hud92] Paul Hudak et al. *Report on the Programming Language Haskell: A Non-Strict, Purely Functional Language*, March 1992. Version 1.2. Also in Sigplan Notices, May 1992.

[Hug89] J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2), 1989.

[jfp94] Special issue on type systems for object-oriented programming. *Journal of Functional Programming*, 4(2), April 1994.

[Läu92] Konstantin Läufer. *Polymorphic Type Inference and Abstract Data Types.* PhD thesis, Department of Computer Science, New York University, New York City, USA, 1992.

[Läu94] Konstantin Läufer. Combining Type Classes and Existential Types. In *Proc. Latin American Informatics Conference (PANEL)*, Mexico, September 1994. ITESM-SEM.

[LO92] Konstantin Läufer and Martin Odersky. An Extension of ML with First-Class Abstract Types. In *Proc. Workshop on ML and its Applications*, San Francisco, June 1992. ACM SIGPLAN.

[Mey88] Bertrand Meyer. *Object-oriented Software Construction.* Prentice Hall, 1988.

[PT94] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–248, April 1994.

[WB89] P. Wadler and S. Blott. How to make *ad hoc* polymorphism less *ad hoc.* In *Proceedings 1989 Symposium Principles of Programming Languages*, pages 60–76, Austin, Texas, 1989.

A The Syntax of Haskell++

The syntax of Haskell is extended as follows:

topdecl	→	**object class** [*objcontext* ⇒] *tycls* [**where** {*objcbody* [;]}]
	|	**object instance** [*context* ⇒] *tycls* (*pat* :: *inst*)
		[**inheriting** $var_1, \ldots, var_m$ **from** *var*]
		[**where** $\{method_1; \ldots; method_n$ [;]}]
objcontext	→	*tycls*
	|	$(tycls_1, \ldots, tycls_n)$
objcbody	→	$objsign_1; \ldots; objsign_n$
objsign	→	*var* :: [*context* ⇒] *type*
method	→	*var* $apat_1 \ldots apat_n$ = *exp* [**where** {*decls* [;]}]

Non-terminals which are left undefined here are defined in the Haskell report[Hud92].

From Hindley-Milner Types to First-Class Structures

Mark P. Jones
Department of Computer Science, University of Nottingham,
University Park, Nottingham NG7 2RD, England.
mpj@cs.nott.ac.uk

Abstract

We describe extensions of the Hindley-Milner type system to support higher-order polymorphism and first-class structures with polymorphic components. The combination of these features results in a 'core language' that rivals the expressiveness of the Standard ML module system in some respects and exceeds it in others.

1 Introduction

The Hindley-Milner type system [13, 28, 8], hereafter referred to as HM, represents a significant and highly influential step in the design and development of programming language type systems. The main reason for this is that it combines the following features in a single framework:

- Type security: soundness results guarantee that well-typed programs cannot 'go wrong'.
- Flexibility: polymorphism allows the use and definition of functions that behave uniformly over all types.
- Type inference: the existence of principal types, and algorithms to compute them, provides a direct way to identify well-typed terms without the use of type annotations.

Note that it is this combination, rather than the individual features themselves, that make HM particularly interesting. For example, there are many different type systems supporting polymorphism of one form or another, while soundness is usually regarded as an essential prerequisite in the design of any type system. HM is also quite attractive from the perspective of language implementors:

- The type inference algorithm is easy to implement and behaves well in practice.
- Polymorphism is easy to implement, packaging function arguments as boxed values—a uniform representation that is independent of the type of values involved. One of the main attractions of this approach is the ease with which it permits true separate compilation.

As a result, HM has been used as a basis for several widely used functional languages including Standard ML (SML) [29] and Haskell [14].

We should also recognize that HM has some significant limitations, including:

- Polymorphism is restricted to the second-order case, i.e. the form of polymorphism used in the polymorphic λ-calculus [38], also known as System F [9, 10]. Higher-order polymorphism of the form provided in F_ω, allowing the specification of functions that behave uniformly over all type constructors is not supported.

- The price that we pay for the convenience of type inference is the inability to define or use functions with polymorphic arguments. By contrast, arguments with polymorphic type can be used in system F and F_ω.

Despite these limitations, and thanks to the use of polymorphism and higher-order functions, HM does permit a limited form of abstraction. For example, the definition of a polymorphic sort procedure may be parameterized by a suitable comparison function. But the need for mechanisms to support large-scale, modular programming have prompted the development of more complex systems, including HM as a 'core'. The best known example of this is the SML module system which has a precise formal definition [29] and has proved to be very useful in practice [4]. Unfortunately, such systems can be quite complicated and may not be fully integrated with the core language. For example, type-theoretic treatments of the SML module system are usually based on the use of dependent types [26, 24, 30]. But it is not easy to discern the role of dependent types in the formal definition of SML [29], which, instead, uses a semantics based on freshly generated tokens or *stamps* to account for the concepts of *sharing* and *generativity*. Recent work by Leroy [22] provides a more type-theoretic treatment of the same concepts but, once again, relies on non-trivial extensions of the underlying dependent type theory. Additional machinery is necessary to extend the SML module system beyond its current definition, for example, to support modules as first-class values [31, 20], higher-order modules [12, 34, 41, 25, 7], polymorphism [19], and *transluscent sums* or *manifest types* [11, 21]. This last item is motivated in part by the desire to admit a form of separate compilation; this is not possible with the current definition of SML [2]. These extended systems are undoubtedly very powerful, but they are also rather complex.

In the remaining sections, we show how relatively modest changes to HM can overcome the limitations above: i.e. to support higher-order polymorphism (Section 2) and to package up polymorphic values in first-class *structures* that can be used as function arguments (Section 3). The resulting system, referred to here as XHM, is of interest in its own right as a study of the boundaries of type inference and also as an implementation language for type and constructor classes [43, 17]. However, in this paper, we focus on the possibility of using structures as the basis for a module system. Unlike the SML module system, we do not make any separation between the core and module languages. For example, structures can be assigned polymorphic types and used as function arguments to implement higher-order modules. Another important difference is that XHM structures do not contain type components (Section 4). The paper ends with a summary of background and related work (Section 5) and with conclusions and suggestions for future work (Section 6).

2 Higher-order HM

The extension of HM to support higher-order polymorphism is surprisingly straightforward. In this section, we review the treatment of higher-order HM suggested by [17], and speculate why this idea has only recently received any significant level of interest.

In standard HM, the type of an object may include variables representing a fixed, but arbitrary type. For example, a polymorphic identity function can be defined using:

$$\begin{array}{lcl} id & :: & a \to a \\ id\ x & = & x \end{array}$$

To emphasize the fact that the variable a represents an arbitrary type, it is common to write the type of id as $\forall a.a \to a$, using explicit universal quantification.

To extend HM to support higher-order polymorphism, we need to allow the use of variables to represent, not just arbitrary types, but also type constructors, etc. Following [17], we use a system of kinds to distinguish between different forms of type constructor, given by the grammar:

$$\begin{array}{rcll} \kappa & ::= & * & \textit{the kind of all (mono)types} \\ & | & \kappa_1 \to \kappa_2 & \textit{function kinds} \end{array}$$

Intuitively, the kind $\kappa_1 \to \kappa_2$ represents constructors which take something of kind κ_1 and return something of kind κ_2. This choice of notation is motivated by Barendregt's description of generalized type systems [3].

For each kind κ, we have a collection of constructors C^κ (including constructor variables α^κ) of kind κ given by:

$$\begin{array}{rcll} C^\kappa & ::= & \chi^\kappa & \textit{constants} \\ & | & \alpha^\kappa & \textit{variables} \\ & | & C^{\kappa' \to \kappa}\ C^{\kappa'} & \textit{applications} \end{array}$$

This corresponds very closely to the way that most type expressions are already written in Haskell. For example, *List a* is an application of the constructor constant *List* to the constructor variable a. In addition, each constructor constant has a corresponding kind. For example, writing $(\to)$ for the function space constructor and $(,)$ for pairing we have:

$$\begin{array}{lcl} \textit{Int},\ \textit{Float},\ () & :: & * \\ \textit{List} & :: & * \to * \\ (\to),\ (,) & :: & * \to * \to * \end{array}$$

The kinds of constructor applications are given by:

$$\frac{C :: \kappa' \to \kappa \qquad C' :: \kappa'}{C\ C' :: \kappa}$$

The task of checking that a type expression is well-formed can now be reformulated as the task of checking that a constructor expression has kind $*$. The apparent mismatch between the explicitly kinded constructor expressions specified above and the implicit kinding used in examples can be resolved by a process of kind inference; i.e. by using standard techniques to infer kinds for user defined constructors without the need for programmer-supplied kind annotations [17].

The set of type schemes is described by the following grammar:

$$\begin{array}{rcll} \tau & ::= & C^* & \textit{monotypes} \\ \sigma & ::= & \forall \alpha^\kappa.\sigma & \textit{polymorphic types} \\ & | & \tau & \end{array}$$

With these definitions in place, we can use the standard notation to specify the typing rules of higher-order HM in Figure 1. Note the use of the symbols τ and σ to restrict the application of certain rules to types or type schemes, respectively. Most of the rules are the same as in HM. In rule $(\forall I)$, the condition $\alpha^\kappa \notin CV(A)$ is needed to ensure that we do not universally quantify over a variable which is constrained by the type assignment A, where the expression $CV(X)$ denotes the set of all constructor variables appearing free in X.

The definition of a type inference algorithm for higher-order HM, and proof of its soundness and completeness, follow almost directly from the corresponding definitions for standard HM. The only complication is in the extension of the unification algorithm to constructors of arbitrary kind. The critical observation there is that there are no non-trivial equivalences between

$$(var) \quad \frac{(x:\sigma) \in A}{A \vdash x : \sigma}$$

$$(\rightarrow E) \quad \frac{A \vdash E : \tau' \rightarrow \tau \quad A \vdash F : \tau'}{A \vdash EF : \tau}$$

$$(\rightarrow I) \quad \frac{A_x, x:\tau' \vdash E : \tau}{A \vdash \lambda x.E : \tau' \rightarrow \tau}$$

$$(let) \quad \frac{A \vdash E : \sigma \quad A_x, x:\sigma \vdash F : \tau}{A \vdash (\textbf{let } x = E \textbf{ in } F) : \tau}$$

$$(\forall E) \quad \frac{A \vdash E : \forall \alpha^\kappa.\sigma \quad C \in C^\kappa}{A \vdash E : [C/\alpha^\kappa]\sigma}$$

$$(\forall I) \quad \frac{A \vdash E : \sigma \quad \alpha^\kappa \notin CV(A)}{A \vdash E : \forall \alpha^\kappa.\sigma}$$

Figure 1: Typing rules for higher-order HM

constructor expressions, and that there is no way to define arbitrary abstractions over constructor variables. This limits the power of the type system quite significantly although it does not have any real impact on the use of XHM to describe modular structure. More importantly, this restriction enables us to avoid the need for higher-order unification which would have resulted in an undecidable type system. To avoid unnecessary distraction from the subject of this paper, we refer the interested reader to [17] for further description and formalization of the topics mentioned here.

Given that the extension of HM to the higher-order case is so straightforward, we might speculate why this idea does not appear to have been explored in any detail elsewhere. One possibility is that the notation for types in some languages forces a first-order view of types. We have been fortunate that the concrete syntax for languages like Haskell encourages us to think of type expressions like $T\ a\ b$ rather than $T(a, b)$.

Another possible explanation is that, by itself, pure higher-order polymorphism seems too general for practical applications. For example, it is hard to think of any useful functions with type $\forall a.\forall m.a \rightarrow m\ a$[1]. The only possibility is the function $\lambda x.\bot$ which, apart from having almost no practical use, can be treated as having the more general type $\forall a.\forall b.a \rightarrow b$ without the need for higher-order polymorphism.

Perhaps the most interesting aspect of higher-order HM is the way that it enables us to define new datatypes with both types and constructors as parameters:

```
data Rec f       = In (f (Rec f))
data ListF a b   = Nil | Cons a b
type List a      = Rec (ListF a)
data StateM m s a = STM (a → m (a, s))
```

The first three examples here can be used to provide a general framework for constructing recursive datatypes and corresponding recursion schemes [27]. The fourth example can be used to describe a parameterized state monad [18, 23].

[1] An obvious way to formalize arguments like this would be to use higher-order analogues of well-known parametricity conditions [39] or free-theorems [42].

The reader may like to check the following kinds for each of the type constructors introduced above.

$$\begin{array}{lll} \mathit{Rec} & :: & (* \to *) \to * \\ \mathit{ListF} & :: & * \to * \to * \\ \mathit{List} & :: & * \to * \\ \mathit{StateM} & :: & (* \to *) \to * \to * \to * \end{array}$$

All of these kinds can be determined automatically without the use of kind annotations. As a final comment, it is worth noting that the implementation of this weak form of higher-order polymorphism is straightforward, and that experience with practical implementations, for example, Gofer, suggests that it is also very natural from a programmer's perspective.

3 Polymorphic values in function arguments

In this section, we describe the second ingredient of XHM: the ability to support function arguments with polymorphic components. Although this may seem to be a rather big step, we start with an example to show that it is in fact already permitted in a system with type or constructor classes where overloaded values are implicitly parameterized by dictionary structures. However, the most important part of this is the use of explicit type information, not the overloading mechanisms involved. From this observation, we move on to a more general discussion of signatures and structures in XHM and to some simple examples to illustrate its use in practice.

3.1 A motivating example using constructor classes

As far as we know, the only other example of work where HM has been extended to higher-order polymorphism is in the study of constructor classes [17] where it is combined with overloading. Mechanisms for overloading provide a form of implicit parameterization that is appropriate when the meaning of a particular symbol is uniquely determined by the types of values that are involved. For example, the treatment of a type constructor as a (category-theoretic) functor in the canonical manner is a good application. But there are also situations where the use of overloading is hard to justify. It would be hard to believe that overloading, by itself, was responsible for the success of higher-order polymorphism in the development of constructor classes.

Closer examination reveals that the real reason for the success of constructor classes is the ability to deal with (implicit) function parameters containing values of polymorphic type. To illustrate this claim, consider the following definitions[2]:

$$\begin{array}{l} \textbf{class } \mathit{Unit}\ m \textbf{ where} \\ \quad \mathit{unit} :: a \to m\ a \\ \\ \mathit{twice} :: \mathit{Unit}\ m \Rightarrow a \to m\ (m\ a) \\ \mathit{twice}\ x = \mathit{unit}\ (\mathit{unit}\ x) \end{array}$$

Note that the definition of *twice* would not be well-typed if *unit* was passed as an explicit parameter as in:

$$\mathit{twice}\ \mathit{unit}\ x = \mathit{unit}\ (\mathit{unit}\ x)$$

In this case, the only possible type for *twice* would be $\forall a.(a \to a) \to (a \to a)$. However, using the dictionary based techniques of Wadler and Blott [43], the constructor class program above

[2]The `Unit` class defined her can be thought of as a cut-down version of the `Monad` class used elsewhere [17, 23] to support the use of monads in functional programming.

is implemented by translating it to obtain the following definitions:

$$\begin{array}{lcl} \textbf{type } \mathit{Unit}\ m & = & \{\, \mathit{unit} :: \forall a.a \to m\ a \,\} \\ \\ \mathit{twice} & :: & \mathit{Unit}\ m \to a \to m\ a \\ \mathit{twice}\ u\ x & = & u.\mathit{unit}\ (u.\mathit{unit}\ x) \end{array}$$

We use the notation $\{\ldots\}$ to represent a record of named components and the familiar notation $r.l$ to denote the extraction of a field l from a record r. The important point in this definition is that the *unit* component of the u record that is passed as an argument to *twice* has a polymorphic type. By instantiating the quantified type variable to a for the inner call of *unit* and to $m\ a$ for the outer call, we obtain the required type. What makes this work is the fact that the *unit* function is declared as a global function with a polymorphic type that allows the type variable a in the example above to be instantiated to different types. It is this additional type information, not the use of overloading, that is important for the work described in this paper. The approach suggested by Wadler and Blott is to translate programs with implicit overloading into a language that makes the use of dictionary structures explicit. In essence, one of the main claims of this paper is that the target of this translation for the system of constructor classes is a useful and powerful language in its own right.

3.2 Records, signatures and structures

To complete the formal description of XHM, we extend the higher-order HM of the previous section to include record types (i.e. additional elements of C^*) of the form:

$$\{\, x_1 :: \sigma_1; \ldots; x_n :: \sigma_n \,\}$$

For convenience, we will often abbreviate record types like this using expressions of the form $\{\, x_i :: \sigma_i \,\}$, where i ranges over some implicit set of indices. We will refer to record types of this form as signatures.

Unlike signatures in the SML module system, signatures in XHM may contain free variables. For example, the variable m appears free in the signature *Unit* m above. Alternatively, we can think of *Unit* as a parameterized signature; in fact, for the purposes of kind inference, *Unit* is treated as having kind $(* \to *) \to *$. On the other hand, SML also permits the declaration of type constructors in signatures, with corresponding definitions in matching structures[3]. We discuss the tradeoffs between signature parameters in XHM and type components in SML more fully below in Section 4.

XHM signatures are also very much like records in SML, with one important difference: the components of an SML record cannot have a polymorphic type. In this way, XHM can be viewed as an attempt to unify the module and record languages of SML. It is also possible that the XHM system can be further extended to support fully polymorphic extensible modules using the approaches suggested by [16, 35], for example. We leave this as a topic for future research. A previous attempt to explain the ML module system using record types has been presented by Aponte [1], but does not make any use of a higher-order type system and differs substantially from the approach described in this paper.

[3] For completeness, we should mention that SML structures may also include definitions of exceptions; we will not address the use of exceptions in this paper.

Structure values will be written using the syntax:

$$\begin{array}{l}\textbf{struct}\\ \quad x_1 \;=\; \langle\text{implementation of } x_1\rangle\\ \qquad\vdots\\ \quad x_n \;=\; \langle\text{implementation of } x_n\rangle\end{array}$$

and will also be abbreviated using expressions of the form $\textbf{struct}\ \{\, x_i = E_i \,\}$, again with i ranging over some implicit set of indices. We have chosen this particular syntax because it fits well with the Haskell layout rule in the prototype implementation; in fact, as the examples below illustrate, we actually allow a more liberal syntax that allows the use of function arguments and pattern matching on the left hand side of equations.

The typing rules for structures are unlikely to cause many surprises. First the rule for constructing structures:

$$\frac{A \vdash E_i : \sigma_i}{A \vdash \textbf{struct}\ \{\, x_i = E_i \,\} : \{\, x_i :: \sigma_i \,\}}$$

It is useful to think of structure expressions as being like local definitions that bind several variables, as in $\textbf{let}\ \{\, x_i = E_i \,\}\ \textbf{in}\ F$ for example, except that the expression over which the bindings are scoped is omitted and the result is the value of the bindings themselves, preserved as a structure value.

A second rule is needed to describe the typing of a component that has been extracted from a structure value. This is also straightforward:

$$\frac{A \vdash E : \{\, x_i :: \sigma_i \,\}}{A \vdash E.x_j : \sigma_j}$$

An alternative, but equivalent, approach is to think of selectors as functions:

$$(\_.x_j) \quad :: \quad \forall v.\{\, x_i :: \sigma_i \,\} \to \sigma_j$$

where v represents the free variables, or parameters, of the signature $\{\, x_i :: \sigma_i \,\}$. If we think of signatures as abstract datatypes, for example, forgetting the definition of *Unit* as a signature type in the previous section and thinking of $Unit\ m$ as some primitive type, then we can write the type of selectors in a more concrete form:

$$(\_.unit) \quad :: \quad \forall m.Unit\ m \to (\forall a.a \to m\ a)$$

Since a does not appear free in the constructor expression $Unit\ m$, we can move the quantifier for a outwards to obtain an equivalent higher-order HM type for the selector:

$$(\_.unit) \quad :: \quad \forall m.\forall a.Unit\ m \to a \to m\ a$$

This gives a strong hint to the implementation of XHM type checking, and also to the proof that XHM programs are sound and have principal type schemes. All that is necessary is sufficient explicit type information to determine which selector type is appropriate in any particular context.

3.3 The use of explicit type information

The need for explicit type information in programs using records will already be familiar to SML programmers; the definition of SML requires that the shape of any record—i.e. a complete

list of its fields—can be determined at compile-time. What we have described here is a natural generalization of this; we require not only the names of all of the fields, but also the types for every field that is referenced. It is fairly easy to arrange for this mechanism to reduce to the SML treatment of records when a polymorphic type has not been specified.

Type annotations are not necessary in many simple examples. For example, the following program type checks without any additional type information:

$$\begin{array}{lcl} f\ x & = & \textbf{struct} \\ & & \quad h\ z = [z] \\ & & \quad u = x \\ v\ y & = & m.h\ (m.h\ m.u) \\ & & \quad \textbf{where}\ m = f\ y \end{array}$$

In this case, we can calculate the following types for the components in the structure value in the definition of f:

$$\begin{array}{lcl} h & :: & \forall t.t \to [t] \\ u & :: & a \end{array}$$

where a is the type of the argument x. Thus:

$$f \ :: \ \forall a.a \to \{\, h :: \forall t.t \to [t];\ u :: a \,\}$$

and it follows that v has type $\forall a.a \to [[a]]$.

In practice, explicit type annotations are typically only required for the definition of recursive or mutually recursive structures (which are not permitted by the SML module system) or for functions that manipulate structure values (corresponding to SML functor definitions where explicit type information is also required in SML, or to higher-order or first-class modules which are not supported by SML). For example, the type signature accompanying the following definition cannot be omitted[4]:

$$\begin{array}{lcl} \mathit{makeUnit} & :: & a \to \mathit{Unit}\ m \to m\ a \\ \mathit{makeUnit}\ x\ u & = & u.\mathit{unit}\ x \end{array}$$

On the other hand, we are free to store values of some type $\mathit{Unit}\ m$ in data structures such as lists and to use many higher order functions, for example $\lambda z.\mathit{map}\ (\mathit{makeUnit}\ z)$, without further type annotations.

Some may question the need for explicit type information in a language that is based on a system like HM, but we do not expect that this will have any significant impact on programmers:

- Some form of explicit type information is already necessary in many languages based on HM. For example, this includes the overloading mechanisms of Haskell and the treatment of records, arithmetic, and structures in SML.
- Explicit type annotations are only required in situations where they would already be required by programs using the SML module system, or in programs that cannot be written with SML modules.
- Despite the fact that it is not necessary, the use of type annotations in implicitly typed languages like ML and Haskell is widely recognized as 'good programming style', and many programmers already routinely include type declarations in their source code. The type assigned to a value serves as a useful form of program documentation. In addition, this gives a simple way to check that the programmer-supplied type signatures, reflecting intentions about the way an object will be used, are consistent with the types obtained by type inference.

[4] Unless some alternative method is used to specify the type of u!

3.4 Simple applications of XHM structures

The examples given above have already demonstrated the use of XHM structures as first-class values. In this section, we give some examples to illustrate other applications of these ideas.

One of the standard examples in texts on abstract datatypes is the definition of a type *cpx* of complex numbers together with a collection of operations for manipulating values of this type. It is often convenient to package these operations together, as described by the signature:

$$\begin{array}{lcll} \textbf{type}\ \mathit{Complex}\ \mathit{cpx} & = & \{\, \mathit{mkCart}, \mathit{mkPolar} & :: \ \mathit{Float} \to \mathit{Float} \to \mathit{cpx}; \\ & & \mathit{re},\ \mathit{im} & :: \ \mathit{cpx} \to \mathit{Float}; \\ & & \mathit{mag},\ \mathit{phase} & :: \ \mathit{cpx} \to \mathit{Float}; \ldots \} \end{array}$$

There are two obvious ways to implement complex numbers using pairs of floating point numbers and choosing either a cartesian or polar representation:

$$\begin{array}{lcll} \mathit{rectCpx} & = & \textbf{struct} & \\ & & \quad \mathit{mkCart}\ x\ y & = \ (x, y) \\ & & \quad \mathit{mkPolar}\ r\ \theta & = \ (r\cos\theta, r\sin\theta) \\ & & \quad \mathit{re}\ (x,\ y) & = \ x \\ & & \quad \ldots & \end{array}$$

$$\begin{array}{lcll} \mathit{polarCpx} & = & \textbf{struct} & \\ & & \quad \mathit{mkCart}\ x\ y & = \ (\mathit{sqrt}\ (x^2 + y^2), \mathit{atan2}\ y\ x) \\ & & \quad \mathit{mkPolar}\ r\ \theta & = \ (r, \theta) \\ & & \quad \mathit{re}\ (r,\ \theta) & = \ r\cos\theta \\ & & \quad \ldots & \end{array}$$

As it stands, the implementation type of both complex number packages is the same and is captured explicitly in the type of each; *Complex* (*Float*, *Float*). Later, in Section 4.3, we will show how to make these types abstract, either completely by using an existential, or partially by giving them names without revealing how they are implemented.

In fact, polymorphism already provides one form of abstraction. To illustrate this, suppose that we want to define a package of complex number arithmetic. We start with a signature for a general arithmetic package, considerably simplified for the purposes of this paper:

$$\begin{array}{lcll} \textbf{type}\ \mathit{Arith}\ a & = & \{\, \mathit{plus} & :: \ a \to a \to a; \\ & & \mathit{neg} & :: \ a \to a; \ldots \} \end{array}$$

Now we can describe the construction of a complex arithmetic package from an arbitrary complex number package using the following function:

$$\begin{array}{lcl} \mathit{compArith} & :: & \mathit{Complex}\ t \to \mathit{Arith}\ t \\ \mathit{compArith}\ c & = & \textbf{struct} \\ & & \quad \mathit{plus}\ z_1\ z_2 \ = \ c.\mathit{mkCart}\ (c.\mathit{re}\ z_1 + c.\mathit{re}\ z_2)\ (c.\mathit{im}\ z_1 + c.\mathit{im}\ z_2) \\ & & \quad \ldots \end{array}$$

Since the same variable, t, appears as a parameter to both the the *Complex* and *Arith* signatures, it is clear that the type of values that the arithmetic operations in the result can be applied to is the same as the type of complex numbers provided as an argument. At the same time, the fact that t is universally quantified ensures that the definition of *compArith* cannot make any assumptions about the implementation of complex numbers.

The definition of *compArith* corresponds to a functor in SML, but there is no need for a special syntax for functor or structure definitions in XHM; the examples above use only the features of the core language which just happen to involve structure values. The use of the same type variable, t, in both argument and result signatures corresponds to a sharing specification[5], often considered one of the most difficult aspects of the SML module system [36]. In this setting, sharing seems more familiar, simply identifying two types by using the same name for each.

For some applications, it is necessary to use type constructors rather than simple types as signature parameters. For example, a specification of queue datatypes, similar in spirit to those in Paulson's textbook [36], might be given by the following signature:

$$\begin{array}{lcll} \textbf{type}\ \mathit{Queue}\ q & = & \{\,\mathit{empty} & :: \quad q\ a; \\ & & \mathit{enq} & :: \quad q\ a \to a \to q\ a; \\ & & \mathit{null} & :: \quad q\ a \to \mathit{Bool}; \\ & & \mathit{hd} & :: \quad q\ a \to a; \\ & & \mathit{deq} & :: \quad q\ a \to q\ a\,\} \end{array}$$

The variable q used here ranges over unary type constructors that correspond to functions mapping types to types, i.e. over constructors of kind $* \to *$. We will also treat the variable a appearing free in the type of each of the queue operators above as if it had been bound by an explicit universal quantifier. For example, the type of the *empty* component in a structure of type $\mathit{Queue}\ q$ is $\forall a.q\ a$.

The following structure describes the standard implementation of queues using lists:

$$\begin{array}{lcll} \mathit{listQueue} & :: & \mathit{Queue\ List} & \\ \mathit{listQueue} & = & \textbf{struct} & \\ & & \quad \mathit{empty} & = \quad \mathit{Nil} \\ & & \quad \mathit{enq}\ n\ q & = \quad q{+\!\!+}[n] \\ & & \quad \mathit{null}\ \mathit{Nil} & = \quad \mathit{True} \\ & & \quad \mathit{null}\ (\mathit{Cons}\ n\ ns) & = \quad \mathit{False} \\ & & \quad \mathit{hd}\ (\mathit{Cons}\ n\ ns) & = \quad n \\ & & \quad \mathit{deq}\ (\mathit{Cons}\ n\ ns) & = \quad ns \end{array}$$

We hope that the reader will have already recognized that the type declaration here is provided for documentation only, and is not required to determine the type of *listQueue*.

4 Structures and signatures for modular programming?

One of the main goals of this section is to address the question of whether XHM can be used as the basis for a module system with signatures and structures corresponding to interfaces and implementations, respectively. The term 'module system' is already widely used in several different ways with meanings including:

- A mechanism to support separate compilation and namespace management.
- A mechanism to enable the decomposition of large programs into small, reusable units in a way that is resistant to small changes in the program.
- A mechanism for defining abstractions.

[5] SML sharing specifications can also be used to specify equalities between structure values. We do not attempt to deal with this in XHM.

In this and following sections, we argue that XHM is consistent with such goals: Separate compilation is ensured by maintaining a clear separation between types and values and program decomposition is supported by the use of higher-order and nested polymorphism. Powerful abstractions can be defined using parameterized structures. On the other hand, XHM does not by itself allow the definition of abstract datatypes, although this can be dealt with using other methods.

We pay particularly close attention to comparisons with the SML module system. The most significant difference between these two systems is the fact that XHM uses parameterized signatures while SML allows type components in signature and structure definitions. It is certainly true that the inclusion of type components can be a very powerful tool; indeed, without careful restrictions, this may prove too powerful, leading to intractability and undecidable type checking. However, we argue that much can be accomplished *without* type components. This, we believe, is also more in the spirit of HM than systems that include or manipulate type components in modules. For example, in Milner's original work [28], types are used as a purely semantic notion, representing subsets of a semantic domain, not as any concrete form of value.

4.1 Lifting type definitions

We start with an important observation, that type definitions in a module can be lifted to the top-level, which helps to explain why it is possible to omit type components from modular structures. For example, consider the following SML fragment:

```
structure s
  = struct
      type T       = Int
      data List a  = Nil | Cons a (List a)
      ...
    end
```

Despite appearances, the type synonym T and the type constructor *List* are *not* local to the definition of s. At any point in the program where s is in scope, these type constructors can be accessed by the names $s.T$ and $s.List$, respectively. Renaming any references to these types and their constructors in the body of s, we can lift these definitions to the top-level, to obtain the following XHM definitions:

```
type s.T       = Int
data s.List a  = s.Nil | s.Cons a (s.List a)
s              = struct
                   ...
```

In effect, all that the datatype definitions in the original SML program accomplish is to define new top-level datatypes in which the type and value constructor names are decorated with the name of the enclosing structure.

In some situations, renaming is not sufficient to allow type definitions to be lifted to the top-level. For example, the *List* datatype in the following functor definition involves a 'free variable'—the type $x.T$, a component of the argument structure x:

```
functor f(x:SIG) : SIG'
  = struct
      data List  =  Nil | Cons x.T List
      ...
    end
```

The solution in this case is to add an extra parameter to the datatype definition before moving it to the top-level, as shown on the right. This is just a form of λ-lifting [15, 37]:

$$\begin{array}{lcl} \textbf{data}\ f.List\ t & = & f.Nil \\ & | & f.Cons\ t\ f.List \\ f & :: & SIG\ t \rightarrow SIG'\ (f.List\ t) \\ f\ x & = & \ldots \end{array}$$

Notice how the parameterized signatures in the type for f capture the relationship between the types involved in the argument x and those involved in the result $f\ x$.

It is also important to mention that, since the form of higher-order polymorphism described in Section 2 allows type constructors to be used as both signature and datatype parameters, the same technique can be used to deal with type constructor components of functor arguments.

4.2 Generativity

Readers with experience of the SML module system will realize that the simple lifting of the datatype f in the functor definition at the end of the previous section is not quite correct. The reason for this is that SML adopts a notion of generativity, producing a new type constructor each time the functor is applied to an argument. Thus two definitions:

$$\begin{array}{lcl} \textbf{structure}\ s_1 & = & f(x) \\ \textbf{structure}\ s_2 & = & f(x) \end{array}$$

will produce structures with incomparable type components. In truth, when we use an SML functor to 'generate' a new datatype, we are in fact constructing a new instance of a fixed datatype, which is then hidden, in essence, by a form of existential quantification. There is no way to express the *List* type produced by applying f to an appropriate argument structure in the notation of SML, so we are forced to package up instantiation of the actual implementation type, $f.List$, and hiding of the resulting type as a single operation.

Lifting type definitions to the top-level allows us to express the type components of the result of functor applications; for the example above, both s_1 and s_2 have type component $f.List\ t$, assuming that x has type $SIG\ t$. We are then free to treat the question of whether we wish to conceal these implementation types as a separate concern. Various methods for achieving this are described below in Section 4.3. Note that higher-order polymorphism is essential, for the general case, to express type components that depend on type constructors of higher kinds in functor arguments.

We see then that, in XHM, there is no need for the same notions of generativity used in SML. Of course, this only affects the static properties of the system. We would normally expect the definitions above to produce two distinct copies of the same structure; these may not be semantically equivalent in a language with side effects, for example, if the generated structures include state components.

4.3 Abstraction

Abstraction—the ability to hide information about the implementation of a module and protect against misuse—is an important feature of module systems that has been mentioned only briefly in the discussion above. The system described in this paper allows us to distinguish between two different forms of abstraction:

- The independence of the implementation of a module from the implementation of its imports.
- The ability to hide details about the type or type constructor parameters in the signature of a structure.

The *compArith* example in Section 3.4 has already illustrated how the first of these can be expressed very elegantly in XHM using polymorphism,

For the second, we can identify at least two different levels of hiding that may be useful. One alternative is to provide a name for a datatype, but to conceal the details of its implementation, such as the constructor functions of an algebraic datatype or the expansion of a type synonym. This is just a matter of scoping, and does not require any changes to the underlying type theory; indeed, it is perhaps best dealt with at the level of compilation units rather than in the core language itself.

For example, we might use a construct:

```
export Cpx, cpx from
  type Cpx  =  (Float, Float)
  cpx       ::  Complex Cpx
  cpx       =   ...
```

The intention here is that the definition of the *cpx* structure and the name of the *Cpx* type are exported, but that the implementation of *Cpx* is not. Outside this definition, a programmer can be sure that *compArith cpx* has type *Arith Cpx*, and that there is no danger of confusing this with any other complex number package that happens to use pairs of floating point numbers as a representation of complex numbers. Several languages support similar constructs, either through explicit namespace management primitives (as in Haskell) or more specific features such as the **abstype** construct in SML. While this approach can be quite useful, it is limited to top-level definitions. For example, we cannot simulate the behaviour of SML functor application outlined above where a new type is generated each time the functor is applied.

A second alternative is to use existential types, concealing not just the implementation, but also the name of an abstract type. We strongly believe that XHM can be extended in a modular fashion to support existential types, for example using dot-notation [5], and the combination of type inference and existential typing that has been explored by Läufer [20]. Note that this can be used to simulate the effects of datatype generativity; for example, if f is assigned a type of the form $SIG\ t \to \exists t'.SIG'\ t'$, then the definitions of structures s_1 and s_2 in the previous section will produce distinct types $s_1.t'$ and $s_2.t'$.

We believe that both of these approaches are useful in their own right. However, neither coincides exactly with the form of abstract datatypes provided by SML; the latter falls somewhere between the two extremes of named and existentially quantified abstract datatypes. It remains as an interesting problem to determine whether there is a modular extension of XHM which provides the same form of abstraction as SML.

4.4 Polymorphic modules

In recent work, Kahrs [19] has shown that it is not possible to define certain kinds of polymorphic module in SML. Fortunately, this problem does not occur with XHM modules. In fact, a direct translation of Kahrs' SML code to our framework already provides the desired form of polymorphism. For reasons of space, we will restrict ourselves to a rather shorter example using

the following SML signature:

```
signature I
= sig
    type T
    id      :: T → T
  end
```

The problem identified by Kahrs is caused by the fact that the type component in any structure matching I must be fixed to some specific type. In particular, it cannot be a variable, and hence it is impossible to define a structure s that matches I such that $s.id$ is the polymorphic identity function.

The corresponding definition in XHM is:

$$\textbf{type } I\ t \quad = \quad \{\, id :: t \to t \,\}$$

But in this case, we can define a structure:

$$s \quad = \quad \textbf{struct} \quad id\ x \quad = \quad x$$

which has type $\forall t.I\ t$ and hence $s.id$ can be used as a polymorphic identity function. We refer the reader to [19] for examples of more useful applications of this form of polymorphism.

4.5 Practical concerns

Practical experience with the SML module system suggests a number of useful features for module system designs and we will take the opportunity for a brief discussion of some of these ideas here.

First, we may be concerned that, in realistic applications, the number of type components in a module may become quite large and that the corresponding parameterized signature would become rather cumbersome and awkward. The easiest way to overcome this problem is to extend the languages of constructors and kinds in the Higher-order HM system to include records (i.e. labelled products) of the form:

$$(t_1 = C^{\kappa_1}, \ldots, t_n = C^{\kappa_n}) :: (t_1 :: \kappa_1, \ldots, t_n :: \kappa_n)$$

and to allow constructor variables ranging over such kinds. This gives the flexibility to package several constructors into a single signature parameter. Apart from reducing the number of parameters that have to be written, this also makes source code more resistant to change if, for example, a new type parameter is added. Type sharing constraints can also be described quite nicely in this framework. One possibility is to use a form of qualified types:

$$someprogram :: (r.x = s.y) \Rightarrow SIG\ r \to SIG'\ s$$

to indicate that the x and y fields of r and s, respectively, are equal. Alternatively, we might adopt a notation based on row variables to express the same constraints:

$$someprogram :: SIG\ (r \mid x = a) \to SIG'\ (s \mid y = a).$$

Subsumption is another useful feature of the SML module system which allows unwanted components of functor arguments to be ignored. Explicit type information is used to support this. We believe that much the same techniques would also be applicable here, although we have not yet attempted to establish this formally. Another interesting idea is to extend work on extensible

records to provide a collection of operators on structures, for example, using $r \setminus l$ to specify the structure obtained by removing the l component from r, or $(r \mid x = v)$ to describe the extension of a structure r with a new component x. This would provide an explicit alternative to the implicit treatment of subsumption.

Finally, it is clear from our earlier discussions that XHM structures have much in common with type classes in languages like Haskell. However, further work is needed to obtain a language design that combines these two features in an elegant and orthogonal manner.

5 Type-theoretic background

To set the contributions of this paper in perspective, this section reviews some of the previous attempts to provide a type-theoretic foundation for modular programming, concentrating in particular, on two of the the most important features of such systems: abstraction and separate compilation.

5.1 Existential typing and abstraction

One way to formalize the process of hiding the implementation of an abstract datatype is to use an existential type [33, 6]:

$$\textbf{type } \mathit{Complex}' \;=\; \exists \mathit{cpx}.\mathit{Complex}\ \mathit{cpx}.$$

Informally, the existential typing indicates that there is a type cpx with the operations listed above defined on it, but prohibits the programmer from making any assumptions about the implementation type. Formally, the properties of existentials are described by the following typing rules, based on standard rules for existential quantifiers in logic:

$$\frac{\Gamma \vdash M : [\tau'/t]\tau}{\Gamma \vdash M : (\exists t.\tau)}$$

This is often described as the introduction rule. Note that the implementation type τ' for the abstract type is discarded and does not appear anywhere in the conclusion.

At the same time, the requirement that N has a polymorphic type in the following elimination rule ensures that we do not make any assumptions about the now hidden implementation type since the definition behaves uniformly for all choices of t:

$$\frac{\Gamma \vdash M : \exists t.\tau \quad \Gamma \vdash N : \forall t.\tau \to \tau' \quad t \notin TV(\tau')}{\Gamma \vdash \textbf{open } M \textbf{ in } N : \tau'}$$

Existential types completely hide the identity of implementation types. For example, the types cpx and cpx' in the body of the following expression cannot be identified, even though they both come from the same term c of type $Complex$:

$$\begin{array}{l}\textbf{open } c \textbf{ in } \Lambda \mathit{cpx}.\lambda x : \mathit{Complex}\ \mathit{cpx}. \\ \quad \textbf{open } c \textbf{ in } \Lambda \mathit{cpx}'.\lambda y : \mathit{Complex}\ \mathit{cpx}'. \\ \quad\quad \ldots\end{array}$$

To emphasize this behaviour, suppose that we define an abstract data type of arithmetic operations, and attempt to reconstruct the *compArith* function from Section 3.4:

$$\begin{array}{lcl} \textbf{type } \mathit{Arith'} & = & \exists a.\mathit{Arith}\ a \\ \\ \mathit{compArith} & :: & \mathit{Complex'} \to \mathit{Arith'} \\ \mathit{compArith}\ c & = & \textbf{open } c \textbf{ in } \Lambda \mathit{cpx}.\lambda p : \mathit{ComplexOps}\ \mathit{cpx}. \\ & & \quad \{\mathit{add}{=}\ldots\} \end{array}$$

Given an implementation c of type *Complex'*, we can use the expression *compArith* c to obtain a package for arithmetic on complex numbers. However, this has no practical use because the typing rules for existentials make it impossible to construct any values to which the *add* and *neg* functions of the resulting package can be applied! The type system does not capture the equivalence of the type of complex numbers used in c with the type of values that can be manipulated by *compArith* c.

An alternative approach to existential typing, using *dot notation* in place of the **open** construct described above, has been investigated by Cardelli and Leroy [5]. The dot notation allows us to identify the implementation types of two packages if they have the 'same name'. This avoids the first problem illustrated above, but not the second. Dot notation is also limited by the unavoidably conservative notions of 'same name' that are needed to ensure decidability of type checking, and is not very well-behaved under simple program transformations.

5.2 Dependent types and separate compilation

Motivated by problems with existential types, similar to those described above, MacQueen [24] argued that dependent types provide a better basis for modular programming. In this framework, structures are represented by pairs $\langle \tau, M \rangle$ containing both a type component τ and a term M whose type may depend on the choice of τ. Structures of this form can be treated as elements of a dependent sum type, described informally by:

$$\Sigma t.f(t) = \{\, \langle \tau, M \rangle \mid M \text{ has type } f(\tau) \,\}.$$

The typing rules for dependent sums are standard (see [26], for example) and can be written in the form:

$$\frac{\Gamma \vdash M : [\tau'/t]\tau}{\Gamma \vdash \langle \tau', M \rangle : (\Sigma t.\tau)} \qquad \frac{\Gamma \vdash M : (\Sigma t.f(t))}{\Gamma \vdash \mathit{snd}\ M : f(\mathit{fst}\ M)}$$

The introduction rule on the left is very similar to the corresponding rule for existentials except that the implementation type, τ', is captured in the structure $\langle \tau', M \rangle$ in the conclusion. The elimination rule on the right indicates that, if M is a structure of type $\Sigma t.f(t)$, then the second component, *snd* M, of M has type $f(\mathit{fst}\ M)$, where *fst* M is the first component of M.

In this setting, the **open** M **in** N construct used in the treatment of existentials can be replaced by the term N (*fst* M) (*snd* M), but this is not *abstraction preserving* in the sense of Mitchell [32]. Informally, dependent sums are more powerful than existentials because of the ability to name the type component *fst* M of a structure M. Of course, this also means that the type component of a structure is no longer abstract.

In a sense, a simple treatment of modules using dependent types is actually *too* powerful for practical systems because it interferes with separate compilation. More precisely, it makes it more difficult to separate compile-time type-checking from run-time evaluation. To illustrate this, we recast the previous definitions of complex number and arithmetic types using dependent

sums to obtain:

$$\begin{array}{lcl} \textbf{type } \mathit{Complex'} & = & \Sigma cpx.\mathit{Complex}\ cpx \\ \textbf{type } \mathit{Arith'} & = & \Sigma a.\mathit{Arith}\ a \\ \\ \mathit{compArith} & :: & \mathit{Complex'} \to \mathit{Arith'} \\ \mathit{compArith}\ c & = & \langle \mathit{fst}\ c, \{\ \mathit{add} = \ldots\}\rangle \end{array}$$

At first glance, this definition suffers from the same problems as the previous version using existentials; the type $\mathit{Complex'} \to \mathit{Arith'}$ does not reflect the fact that the type components of the argument and result structures are the same. However, this information can be obtained by carrying out a limited degree of evaluation during type checking. For example, if $c = \langle \tau, M\rangle$, then:

$$\mathit{fst}\ (\mathit{compArith}\ c) = \mathit{fst}\ \langle \tau, \{\ \mathit{add} = \ldots\}\rangle = \tau.$$

To ensure that static type checking is possible, it is important to distinguish compile-time evaluation of this kind from arbitrary run-time execution of a program. Unfortunately, treating a functor as a function of type $(\Sigma s.f(s)) \to (\Sigma t.g(t))$ does not reflect this separation; in general, a type of this form may include elements in which the type component of the result depends on the value component of the argument. As an alternative, Harper, Mitchell and Moggi [12, 34] have shown that a suitable *phase distinction* can be established by modelling functors from $\Sigma s.f(s)$ to $\Sigma t.g(t)$ as values of type $\Sigma h.(\forall s.f(s) \to g(h(s)))$ where h ranges over functions from types to types, corresponding to the compile-time part of the functor.

As the example above shows, it is sometimes necessary to inspect the implementation of a structure to find the value of a type component. Not surprisingly, this means that it is not possible to provide true separate compilation for SML [2]. Even the 'smartest recompilation' scheme proposed by Shao and Appel [40] does not permit true separate compilation because it delays some type checking, and hence the detection of some type errors, to link-time.

The need for type sharing constraints in functor definitions is, in fact, motivated by similar problems; since it is impossible to evaluate the formal parameters of a functor, we must instead supply the required identities between type components using explicit sharing equations. Further extensions to the basic theory of dependent types are needed to deal with this, and other ideas including generativity, polymorphism, abstraction, higher-order modules, and modules as first-class values.

5.3 Translucent sums and manifest types

Recent proposals for *translucent sums* by Harper and Lillibridge [11] and *manifest types* by Leroy [21] provide a compromise between existential typing and dependent sums, allowing the programmer to include additional type information in the signature for a structure. For these systems, we use an introduction rule of the form:

$$\frac{\Gamma \vdash M : [\tau'/t]\tau}{\Gamma \vdash M : (\exists t = \tau'.\tau)}$$

Notice that, unlike the previous cases, the implementation type τ' appears in the inferred type although this can be hidden by coercing it to a standard existential type:

$$\frac{\Gamma \vdash M : (\exists t = \tau'.\tau)}{\Gamma \vdash M : (\exists t.\tau)}$$

These systems provide better support for abstraction and separate compilation than Standard ML. However, the underlying theories are quite complex and unfamiliar, relying on the use of dependent types.

5.4 Comparison with XHM

For all of the examples described above, the type of a module, package or structure is given by an expression of the form $Qt.f(t)$ for some parameterized record type $f(t)$ and some quantifier Q. Most of the problems described above are caused by the fact that these quantifiers can be 'overly protective', limiting the ability to propagate type information. In the previous sections of this paper, we have shown how to use record types of the form $f(t)$ as signatures, treating quantifiers as a separate issue.

6 Conclusion

We have presented an extension of the Hindley-Milner type system that provides:

- Support for higher-order polymorphism.
- Support for function arguments with polymorphic components, sometimes requiring explicit type information to guide the main type inference process.
- A clear separation between static and dynamic semantics.

This leads to a module system in which:

- Structures are first-class values.
- Higher-order modules (i.e. first-class functors) are admitted. Leroy [21] gives several examples of higher-order modules, some of which can only by typed using his system of manifest types, while others require different higher-order extensions of SML [25]. All of these examples, including those in [25], can be typed in XHM.
- Polymorphic modules and structures may be defined.
- True separate compilation is possible: the interface to a separately compiled structure is completely determined by its signature, so there is no need to inspect its implementation.
- Since the module language is based on HM, we believe that it will be easy to use, particularly for programmers that are already familiar with the core languages of Standard ML, Haskell or similar languages.
- We avoid the need for some of the more complicated aspects of the Standard ML type system such as generativity and sharing.

By contrast, none of these is possible with the SML module system without, often substantial, modification. One of the main topics for future research is to investigate the role of implicit subsumption; i.e. the ability to discard elements from a structure as a result of signature matching in SML. We believe that this can be accomplished using a simple form of subtyping, guided by type annotations, or otherwise by extending XHM with an mechanism for controlling the set of bindings that are exported from a structure.

We are currently working on a practical implementation of the ideas presented in this paper for a Haskell-like language. The main goal of this work is to provide the functionality of SML-style modules while preserving the basic character of Haskell, including type classes, side-effect free, call-by-name/lazy semantics, implicit mutual recursion, and so forth. We intend to use our prototype to investigate the use of XHM modules in realistic applications, and to assess how well the ideas described in this paper fare in a practical module system.

Acknowledgements

Many of the ideas presented in this paper were developed while the author was a member of the Department of Computer Science, Yale University, supported in part by a grant from ARPA, contract number N00014-91-J-4043.
My thanks to Paul Hudak, Sheng Liang and, in particular, Dan Rabin and Xavier Leroy for their valuable comments and suggestions during the development of the ideas presented in this paper.

References

[1] María Virginia Aponte. Extending record typing to type parametric modules with sharing. In *Proceedings 20th Symposium on Principles of Programming Languages.* ACM, January 1993.

[2] Andrew W. Appel and David B. MacQueen. Separate compilation for Standard ML. In *Conference on Programming Language Design and Implementation*, Orlando, FL, June 1994.

[3] H. Barendregt. Introduction to generalised type systems. *Journal of functional programming*, 1, April 1991.

[4] Edoardo Biagioni, Robert Harper, Peter Lee, and Brian G. Milnes. Signatures for a network protocol stack: A systems application of Standard ML. In *Proceedings of the 1994 ACM conference on Lisp and Functional Programming*, Orlando, FL, June 1994.

[5] Luca Cardelli and Xavier Leroy. Abstract types and the dot notation. Technical Report report 56, DEC SRC, 1990.

[6] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4), December 1985.

[7] Pierre Crégut and David MacQueen. An implementation of higher-order functors. In *Record of the 1994 ACM SIGPLAN Workshop on ML and its Applications*, Orlando, FL, June 1994.

[8] L. Damas and R. Milner. Principal type schemes for functional programs. In *9th Annual ACM Symposium on Principles of Programming languages*, pages 207–212, Albuquerque, N.M., January 1982.

[9] J.-Y. Girard. Une extension de l'interprétation de Gödel à l'analyse et son application à l'élimination des coupures dans l'analyse et la théorie de types. In Fenstad, editor, *Proceedings of the Scandanavian logic symposium.* North Holland, 1971.

[10] Jean-Yves Girard. The system F of variable types, 15 years later. In Gérard Huet, editor, *Logical Foundations of Functional Programming*, chapter 7, pages 87–126. Addison Wesley, 1990.

[11] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Conference record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 123–137, Portland, OR, January 1994.

[12] Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *Conference record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 341–354, San Francisco, CA, January 1990.

[13] R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, December 1969.

[14] P. Hudak, S. Peyton Jones, and P. Wadler (editors). Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.2). *ACM SIGPLAN Notices*, 27(5), May 1992.

[15] T. Johnsson. Lambda lifting: transforming programs to recursive equations. In Jouannaud, editor, *Proceedings of the IFIP conference on Functional Programming Languages and Computer Architecture*, pages 190–205, New York, 1985. Springer-Verlag. Lecture Notes in Computer Science, 201.

[16] Mark P. Jones. *Qualified Types: Theory and Practice.* PhD thesis, Programming Research Group, Oxford University Computing Laboratory, July 1992. Published by Cambridge University Press, November 1994.

[17] Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. In *FPCA '93: Conference on Functional Programming Languages and Computer Architecture, Copenhagen, Denmark*, New York, June 1993. ACM Press.

[18] M.P. Jones and L. Duponcheel. Composing monads. Research Report YALEU/DCS/RR-1004, Yale University, New Haven, Connecticut, USA, December 1993.

[19] Stefan Kahrs. First-class polymorphism for ml. In D. Sannella, editor, *Programming languages and systems - ESOP '94*, New York, April 1994. Springer-Verlag. Lecture Notes in Computer Science, 788.

[20] Konstantin Läufer and Martin Odersky. An extension of ML with first-class abstract types. In *ACM SIGPLAN Workshop on ML and its Applications*, San Francisco, June 1992.

[21] Xavier Leroy. Manifest types, modules and separate compilation. In *Conference record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 109–122, Portland, OR, January 1994.

[22] Xavier Leroy. A syntactic theory of type generativity and sharing. In *Record of the 1994 ACM SIGPLAN Workshop on ML and its Applications*, Orlando, FL, June 1994.

[23] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Conference record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Francisco, CA, January 1995.

[24] David MacQueen. Using dependent types to express modular structure. In *13th Annual ACM Symposium on Principles of Programming languages*, pages 277–286, St. Petersburg Beach, F.L., January 1986.

[25] David B. MacQueen and Mads Tofte. A semantics for higher-order functors. In D. Sannella, editor, *Programming languages and systems - ESOP '94*, New York, April 1994. Springer-Verlag. Lecture Notes in Computer Science, 788.

[26] Per Martin-Löf. Constructive mathematics and computer programming. In *Logic, Methodology and Philosophy of Science, VI.* North Holland, Amsterdam, 1982.

[27] Erik Meijer and Mark P. Jones. Gofer goes bananas. In preparation, 1994.

[28] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3), 1978.

[29] Robin Milner, Mads Tofte, and Robert Harper. *The definition of Standard ML.* The MIT Press, 1990.

[30] John Mitchell and Robert Harper. The essence of ml. In *Fiftheenth ACM Symposium on Principles of Programming Languages*, San Diego, CA, January 1988.

[31] John Mitchell, Sigurd Meldal, and Neel Madhav. An extension of Standard ML modules with subtyping and inheritance. In *Conference record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, Orlando, FL, January 1991.

[32] John C. Mitchell. On abstraction and the expressive power of programming languages. In T. Ito and A.R. Meyer, editors, *Theoretical Aspects of Computer Software*, New York, September 1991. Springer-Verlag. Lecture Notes in Computer Science, 526.

[33] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, July 1988.

[34] Eugenio Moggi. A category-theoretic account of program modules. In *Summer conference on category theory and computer science*, pages 101–117, New York, 1989. Springer-Verlag. Lecture Notes in Computer Science, 389.

[35] Atsushi Ohori. A compilation method for ml-style polymorphic record calculi. In *Proceedings 19th Symposium on Principles of Programming Languages.* ACM, January 1992.

[36] L.C. Paulson. *ML for the working programmer.* Cambridge University Press, 1991.

[37] S.L. Peyton Jones. *The implementation of functional programming languages.* Prentice Hall, 1987.

[38] J.C. Reynolds. Towards a theory of type structure. In *Paris colloquium on programming*, New York, 1974. Springer-Verlag. Lecture Notes in Computer Science, 19.

[39] J.C. Reynolds. Types, abstraction, and parametric polymorphism. In R.E.A. Mason, editor, *Information Processing 83*, Amsterdam, 1983. North-Holland.

[40] Z. Shao and A. Appel. Smartest recompilation. In *Proceedings 20th Symposium on Principles of Programming Languages.* ACM, January 1993.

[41] Mads Tofte. Principal signatures for higher-order program modules. In *Conference record of the Nineteenth annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, January 1992.

[42] P. Wadler. Theorems for free! In *The Fourth International Conference on Functional Programming Languages and Computer Architecture, FPCA '89*, Imperial College, London, September 1989.

[43] P. Wadler and S. Blott. How to make *ad hoc* polymorphism less *ad hoc*. In *Proceedings of 16th ACM Symposium on Principles of Programming Languages*, pages 60–76, Jan 1989.

Data Compression in Haskell with Imperative Extensions A Case Study

Peter Thiemann*
Universität Tübingen, Germany

1 Introduction

Many algorithms have elegant implementations in functional programming languages. However, for certain problems performance degrades since variables in the imperative sense (and hence side-effects) are absent [12]. This is especially true for algorithms which destructively change an internal state. A well-known example for such an algorithm is depth-first-search in graphs [14]. Its complexity ($O(v+e)$, where v is the number of vertices and e the number of edges of the graph) hinges on the availability of a *mutable* data structure (ie a boolean vector), where every mutation takes constant time. Data structures of the latter kind are simply not available in current functional languages and must therefore be simulated using other (immutable) data structures, eg balanced trees[1]. In terms of the complexity we obtain an additional factor $O(\log v)$ and depth-first search is rendered with a time-complexity of $O(v \log v + e)$.

A new programming style called *imperative functional programming* [11, 6, 7] integrates I/O-operations as well as lazy computations on mutable variables and arrays in a lazy functional language. This integration keeps all the good properties of pure functional programming languages intact, eg freedom of side-effects, simple program transformation, and verification (just to name a few). King and Launchbury [5] give an impressive demonstration of this with several graph algorithms based on depth-first search. Also, the potential of parallel execution inherent in functional languages need not be hampered if imperative features are used [16].

The current case study is an attempt to assess imperative functional programming with respect to a real application. The real application in our case is the LZW algorithm for data compression (Lempel, Ziv, and Welch [18, 17]). It is really an every-day application, since watching pictures in the (formerly) popular GIF-format implies decompression using exactly this algorithm. The well-known program compress employs this algorithm as well.

The performance of the LZW-algorithm depends crucially on global mutable state, which may be a hash table or a trie (see below). Furthermore, it depends on the availability of efficient bit operation in the language, since—in theory—the output of the algorithm is a sequence of bits, which must be transformed into a sequence of characters so as to be written to a file. Last not least, I/O-throughput plays an important rôle as well.

For our evaluation we have implemented different variants of the LZW algorithm in Haskell [3] making use of the imperative extensions as supported by the Glasgow Haskell compiler [9]. For comparison we have used programs written in "pure" Haskell and C [13]. The expressiveness of Haskell is demonstrated by the fact that the current text is at the same time a complete working Haskell program, which implements the LZW algorithm. The following paragraph is a sacrifice towards this aim[2].

*Universität Tübingen, Sand 13, D-72076 Tübingen, Germany, E-mail: thiemann@informatik.uni-tuebingen.de

[1] The language Haskell [3] has arrays, but they are monolithic (all elements are defined on construction of the array) and immutable (all elements remain constant through the whole life-time of the array).

[2] Compile with `ghc-0.23 -cpp -fglasgow-exts lzw-trie.lhs -o lzw-trie`.

```
> module Main
> where
> import PreludeGlaST hiding (seqST) renaming
>        (unsafeInterleaveST to interleaveST,
>         thenStrictlyST to thenST,
>         thenST to thenLazilyST,
>         unsafeFreezeArray to freezeArray,
>         _MutableArray to MutArr,
>         _ST to ST)

> seqST         :: [ST s ()] -> ST s ()
> seqST []      =  returnST ()
> seqST (m: ms) =  m `thenST` \ () -> seqST ms

#define runST _runST
```

The remainder of the text is structured as follows. Section 2 briefly introduces imperative functional programming in Haskell, directed towards operations on mutable arrays. Section 3 explains the implementation of the LZW algorithm using the imperative extensions. Section 4 introduces a new primitive operation, which allows the delayed creation of arrays and Sec. 5 shows the implementation of "staged arrays". Section 6 gives runtimes and memory usage in comparison to other implementations and discusses problems in programming with lazy functional languages that aggravate in connection with the imperative extensions. Following the conclusion (Sec. 9) there are some appendices which make the program runnable. They define the main program, some conversion functions, and the data representation.

2 Imperative Extensions of Haskell

The imperative extensions of Haskell are built on *state transformers* [6]. A state transformer of type `ST s a` is a function which maps a state of type `s` to a pair consisting of a result of type `a` and a new state:

```
type ST s a = s -> (a, s)
```

Now, there are primitive transformers which are necessary to build and compose transformers out of basic transformers. The transformer `returnST x` leaves the state `s` alone and delivers the result `x`:

```
returnST :: a -> ST s a
returnST x = \s -> (x, s)
```

Next, it must be possible to combine two transformers to form a new transformer. The result of the first transformer shall be the input of the second transformer. The required function is:

```
thenST :: ST s a -> (a -> ST s b) -> ST s b
thenST m f = \s -> let (x', s') = m s in f x' s'
```

The interesting fact with the functions `returnST` and `thenST` is that the state `s` is threaded through the whole computation without being duplicated or discarded. This way, the sequence in which primitive state transformers are executed is fixed.[3]

Now we can add basic operations on mutable arrays as state transformers:

```
type MutArr s ix value          -- abstract
```

An object of type `MutArr s a b` is a mutable array which is only usable in state `s`, which is indexed by values of type `ix`, and the elements of which all have type `b`. Only the following operations are defined on `MutArr`:

```
newArray   :: (ix, ix) -> value -> ST s (MutArr s ix value)
readArray  :: MutArr s ix value -> ix -> ST s value
writeArray :: MutArr s ix value -> ix -> value -> ST s ()
```

[3] More precisely: the execution sequence is fixed only for those transformers which are strict in the propagated state.

Here, `newArray (low, high) init` is a transformer which yields a newly created array with index range `low` through `high` inclusive, with all elements set to `init`. The transformer `readArray arr i` yields the element `i` of array `arr`. Finally, `writeArray arr i x` is a transformer which overwrites the `i`th element of array `arr` with new value `x`. This last operation does not copy the array, but changes it in-situ "destructively", as common in imperative languages.

In contrast to imperative languages, in the first place a state transformer is a data object like any other, ie evaluation of `newArray (low, high) init` yields a function which only creates a new array when it is finally executed. Conceptually, the whole "state-based computation" is first constructed and then executed. By virtue of lazy evaluation the construction and evaluation really occur in an interleaved manner. This is the well-known "coroutining effect" of lazy evaluation [4].

A transformer is (really and finally) executed when it is subjected to the function `runST`, which accepts a transformer, executes it, and returns its final result while discarding the final state. It serves to embed state-based computations in pure functional computations. The desired encapsulation is guaranteed by giving `runST` a special type [6, 15]. Simplified, every execution of `runST` generates a fresh state with a fresh type that does not occur anywhere else. Thus, mutable objects that are created and manipulated in a specific state cannot be manipulated in any other state, they are tied to their creating state by the state parameter in their type. Any program that tries to compromise this encapsulation is not type correct and hence rejected by the type checker.

As already said above, the sequence in which the operations are executed is fixed (by their order in the program text). But it is not required that all of the operations must be executed. Only those read and write operations on mutable arrays are actually performed that are necessary to produce the final result. It is out of the scope of this paper to provide more detail on how this is achieved. A comprehensive account is found in [7].

3 The Algorithm of Lempel, Ziv, and Welsh

The LZW algorithm comprises a function for coding text and one for decoding it. Coding compresses a sequence of characters to a sequence of code words, while decoding reverses this process.

```
> type Byte = Char
> byteRange = (0, 255)

> type Code = Int
> initialCode = snd byteRange + 1
> maxCode     = 2 ^ 12 - 1
> codeRange   = (0, maxCode)
```

Input and output characters both have type `Byte`. Its range is specified with `byteRange`.

`Code` is the type of code words. Every code word represents a sequence of characters. `initialCode` is the lowest code which is ever assigned by the algorithm. It represents the first to characters of the input. The code words `0` through `initialCode-1` are reserved for the single character sequences. `maxCode` is the maximum code which is assigned by the algorithm. The current implementation confines the maximum code to fit into 12 bits.

3.1 Coding

```
> data Table s    = Empty | Table Code (VecTable s)
> type VecTable s = EncodeArray s (Table s)
```

During the process of coding, a trie is constructed, ie a tree where the nodes are labeled with code words and the arcs are labeled with input characters. Every path from the root to a node corresponds to a sequence of characters which already occurred in the input. The label at the node reached by some sequence of characters is its encoding. The tree expands dynamically as coding proceeds and new sequences are encounters. Hence, the tree data structure should be mutable.

Every node of this tree is an element of the type `Table s`[4]. It is either `Empty` (if the sequence that leads to this node is not yet present in the tree) or it is of the form `Table code vector`, ie a pair of a code word `code` and a `vector` of successor nodes. The latter is a mutable array of type `VecTable s` where the i^{th} element determines the successor node which is reached through the arc labeled with i.

First, the trie must be initialized with the entries and code for every possible one-character-sequence. The codes of these sequences are in the "forbidden" range `(0, initialCode-1)`. The resulting root table is of type `VecTable s` as it is never empty.

```
> initLZWtable =
>       newEncodeArray byteRange Empty 'thenST' \ root ->
>       seqST [ newEncodeArrayLazily byteRange Empty 'thenST' \ branch ->
>               writeEncodeArray root byte (Table byte branch)
>               | byte <- [fst byteRange .. snd byteRange]]
>       'thenST' \_ -> returnST root
```

In line 2 a new array is created with index range `byteRange`. It represents the root of the trie. Lines 3–5 allocate the empty successor arrays for every `byte` in `byteRange`. The result is the root of the trie.

The construction of the vectors of successor nodes is an interesting point: if all arrays would be allocated immediately, the initialization process alone would devour space for the 256 elements of the root node and for the $256 \cdot 256 = 65536$ elements of the successor vectors for the immediate successors of the root. Obviously, this is a waste of resources since often (eg when coding a text file) not all possible characters occur in an input file. This is remedied through the *delayed construction* of the branch-arrays using `newEncodeArrayLazily` (cf. Sec. 4). These arrays are only created when they are first accessed.

A further reduction of space usage should be possible through the use of *staged arrays*. A staged array is an array with one (or more) stages of indirection. The outer levels of arrays only contain references to sub-arrays which contain a subrange of the index range and are only allocated on demand, and so on (see Sec. 5).

The following function `enter` does the real coding work. It accepts a sequence of characters of type `[Byte]`, the code of the already processed sequence `code`, the current position in the trie `vectable`, the root of the trie `rootvec`, and the next available code `nextcode`.

```
> enter :: [Byte] -> Code -> VecTable s -> VecTable s -> Code -> ST s ([Int])
> enter [] code vectable rootvec nextcode =
>       returnST [code]
> enter (byte: bytes) code vectable rootvec nextcode =
>       readEncodeArray vectable (ord byte) 'thenST' \ table ->
>       case table of
>       Empty ->
>           (if nextcode <= maxCode then
>               newEncodeArrayLazily byteRange Empty 'thenST' \ newtable ->
>               writeEncodeArray vectable (ord byte) (Table nextcode newtable)
>            else
>               returnST ())
>           'thenST' \ () -> enter (byte: bytes) 0 rootvec rootvec (nextcode+1)
>           'thenST' \ morecode -> returnST (code: morecode)
>       Table code vectable ->
>           enter bytes code vectable rootvec nextcode
```

In order to code the next character `byte` of the input sequence, it is first checked, whether there is a corresponding successor node in the trie (L5). In this case, the remaining input is coded starting from the successor node (L15,16). Otherwise (L7–14) it is checked whether there are still code words available. If this is the case, a new node is (lazily) created (L9,10) and written to the successor vector of its predecessor, together with its code. Nothing happens in the other case. In both cases coding continues with the remaining input *including the character just processed* starting over from the root. The result is the code of the predecessor node followed by the coding of the remaining input.

If the input is depleted the code of the previously visited node is returned (L2,3).

[4]The type parameter `s` is only present for technical reasons.

The function encode transforms a sequence of character into a state transformer, which returns a list of code words. It first initializes the trie and then starts enter with the input sequence bytes on the root of the trie.

```
> encode :: [Byte] -> ST s [Code]
> encode bytes =
>       initLZWtable 'thenST' \ rootvec ->
>       enter bytes 0 rootvec rootvec initialCode
```

3.2 Decoding

In order to decode a sequence of code words, a mapping from code words to sequences of characters must be available. This mapping is constructed during the decoding process, similar to the trie construction during coding. We realize the mapping using a mutable array with index range codeRange. This array is first initialized with the sequences of length one.

```
> initLZWdecode =
>       newDecodeArray maxCode "" 'thenST' \ dtable ->
>       seqST [writeDecodeArray dtable byte [chr byte]
>              | byte <- [fst byteRange .. snd byteRange]]
>       'thenST' \_ -> returnST dtable
```

Decoding (decode0) is simple in principle: every incoming code word is looked up in the decode array and the character string str that is found there is returned. Simultaneously, a new entry is deposited in the decode array. Its character sequence is determined from str and the first character of the decoding of the next code word.

The only problem is that under certain circumstances a code word may occur in the input before its value is entered in the decode array. This case occurs if during compression the trie contains an entry for the character sequence *cw* and the next following input starts with *cwcwc* (*c* is a single character, *w* is an arbitrary sequence of characters). In this case, the code for *cw* is emitted upon reading the second *c*, and the code for *cwc* upon reading the third *c*. At decoding time, the latter code cannot yet be in the table. Therefore, the sequence of characters returned from decoding the previous code word must be emitted, extended at the end by its first character.

```
> decode0 dtable nextcode laststr (code: rest) =
>       (if code >= nextcode then
>          returnST (laststr ++ [head laststr])
>        else
>          readDecodeArray dtable code) 'thenST' \ str@(byte: _) ->
>       (if nextcode > maxCode then
>          returnST ()
>        else
>          writeDecodeArray dtable nextcode (laststr ++ [byte]))
>       'thenST' \_ ->
>       decode0 dtable (nextcode + 1) str rest 'thenST' \ showRest ->
>       returnST (showString str . showRest)
> decode0 dtable nextcode laststr [] =
>       returnST id

> decode00 dtable (code: rest) =
>       readDecodeArray dtable code 'thenST' \ str ->
>       decode0 dtable initialCode str rest 'thenST' \ showRest ->
>       returnST (showString str . showRest)

> decode :: [Byte] -> ST s ShowS
> decode codestring =
>       initLZWdecode 'thenST' \ dtable ->
>       decode00 dtable (ercode codestring)
```

4 Lazy Array Allocation

When the trie is constructed it is useful to allocate the arrays at the single nodes lazily. That is, they should only be allocated if they are really needed. Since a sequence of characters which has been entered in the trie may never again occur in the remaining input (and be read starting from the root of the trie), chances are not bad that the vector of the successor nodes is never used. Therefore, every such vector of successor nodes is allocated lazily.

The provided transformer `newArray` does not have this property. Whenever it is executed the array is immediately allocated. To implement `newArrayLazily` we need a new primitive transformer, which accepts a transformer and runs its *independent* from the rest of the computation. This transformer `interleaveST` of type `ST s a -> ST s a` allows us to escape from the strict sequentialization of a state thread and to start an independent sequential computation. As it is possible to introduce side-effects using `interleaveST`, it is not *safe*[5], but sometimes extremely useful.

The application which we demonstrate here is free of side-effects, since only the allocation of the array is delayed. Every read or write operation on an array is strict in the array. This strictness forces the allocation of the array before the actual access happens.

```
> newArrLazily :: (Int, Int) -> b -> ST s (MutArr s Int b)
> newArrLazily bounds init = interleaveST (newArray bounds init)
```

The correct implementation of such a data structure is non-trivial in an imperative programming language. In Haskell, the delayed allocation of arrays can be realized with *one* local change in the program text. In many other languages non-local changes are necessary to accomplish.

5 Staged Arrays

When an ASCII text file is compressed, it contains about 100 different characters. Therefore, every successor vector in the trie is at least 50% empty. To put an end to this waste of space and time[6] it appears sensible to augment the vectors with an indirection stage. On the outermost level there is an arrays of mutable arrays. Each of these serves a specific interval of the entire index range. Since more than half of the arrays are never used, they should be allocated lazily: the array is allocated on first use; if it is never used it does not consume memory[7].

The desired data structure, the *staged array*, is represented by a pair consisting of a standard Haskell-array (the array of indirections, where the elements are mutable arrays) and the span of the intervals. For simplicity, all array are indexed with integers.

To allocate a staged array the indirection array is first created as a mutable array, which is then filled with delayed arrays, and finally transformed into a Haskell array (using `freezeArray`). The latter operation must copy its argument array in general, since side-effects on the Haskell array are otherwise possible. In the present case, there is no further live reference to the mutable array (`stage1` does not occur free in `\arr -> ...`), hence we need not insist on copying the array. It remains to answer the question why the indirection array is not simply left as a mutable array. The reason is the following operation `createStagedArray`, a delayed variant `newStagedArray`:

```
> type StagedArray s b = (Array Int (MutArr s Int b), Int)

> newStagedArray :: Int -> (Int, Int) -> b -> ST s (StagedArray s b)
> newStagedArray chunkSize (lo, hi) init =
>       let lochunk = lo `div` chunkSize
>           hichunk = hi `div` chunkSize
>           dummy   = error "uninitialized staged array"
>       in  newArray (lochunk, hichunk) dummy `thenST` \ stage1 ->
>           seqST [newArrLazily (0, chunkSize - 1) init `thenST` \ realArray ->
>                  writeArray stage1 chunk realArray
```

[5] Indeed, Glasgow Haskell calls it `unsafeInterleaveST`.

[6] The arrays must be initialized.

[7] Unfortunately, its suspension consumes some amount of memory.

```
>                 | chunk <- [lochunk .. hichunk]]
>           'thenST' \_ ->
>           freezeArray stage1 'thenST' \ arr ->
>           returnST (arr, chunkSize)


> createStagedArray :: Int -> (Int, Int) -> b -> ST s (StagedArray s b)
> createStagedArray c s i =
>       interleaveST (newStagedArray c s i)
```

Since the transformer `interleaveST` generates a new independent thread of computation, `newStagedArray` only executes those operations that are necessary to compute the result (the argument on `returnST`). If this was just `returnST (stage1, chunkSize)` only the array of indirections would ever be created, the delayed arrays would never be written, and the first access to any of these would yield a runtime error[8]. In contrast, `freezeArray` only returns a result after all previous transformers have executed and delivered their results[9].

Reading and writing of staged arrays is less interesting and only included for completeness.

```
> readStagedArray :: StagedArray s b -> Int -> ST s b
> readStagedArray (stage1, chunkSize) i =
>       let chunk  = i 'div' chunkSize
>           offset = i 'mod' chunkSize
>       in  readArray (stage1 ! chunk) offset
>
> writeStagedArray :: StagedArray s b -> Int -> b -> ST s ()
> writeStagedArray (stage1, chunkSize) i x =
>       let chunk  = i 'div' chunkSize
>           offset = i 'mod' chunkSize
>       in  writeArray (stage1 ! chunk) offset x
```

6 Assessment and Comparison

The comparison described below refers to the following programs:

compress	compress from the SUNOS 4.1.3 distribution.
lzwc	A simple implementation in C using tries (as balanced trees) [13].
hcz	The corresponding program in (pure) Haskell [13].
lzwh	An improved version of hcz due to John v. Groningen [8].
lzw-trie	The program described in the current text.
lzw-hash	Another version of the current program which employs a hash table instead of a trie. The hash table is implemented as a mutable array. The elements are overflow lists of pairs of key and data.
lzw-it1	Version using an imperative version (with mutable variables) of the data structure of lzwh.
lzw-it2	The same, but implemented with mutable arrays.

For experimental evaluation we have used some files from the "Standard Calgary Text Compression Corpus" [1, 2] (the same as in [13]). Here are their characteristics:

name	size	contents
paper1	53161	text: scientific
bib	111261	text: bibliography (REFER-format)
geo	102400	binary: geophysical data
book1	768771	text: novel—fiction

[8] From the vivid description you can imagine that we found out the hard way.

[9] Another solution would be to use the strict variant `returnStrictlyST` [7] in place of the original `returnST`. It also forces all preceding operations to execute before it delivers any return value.

All times are "user times" and have been measured using the built-in command `time` of *tcsh*, version 6.04.00, on a SPARCstation 10 with 32MB main memory (see Fig. 1). Each of these are median values over 10 runs of the resp. program. The fractions given (speedup/slowdown factors) are computed from the numbers for "paper1". All Haskell-programs have been compiled using the Glasgow-Haskell-Compiler, version 0.23, the C-program with `gcc`, version 2.6.3, all with optimization (option `-O`). Furthermore, in the lines marked

program \ input	paper1	bib	geo	x/lzwh	lzwh/x
compress	0.16	0.30	0.38	0.09	10.68
lzwc	0.18	0.31	—[a]	0.07	13.15
hcz	30.50	53.77	73.36	17.15	0.05
lzwh	1.71	3.30	4.30	1.00	1.00
lzw-trie	2.02	3.60	7.27	1.18	0.84
lzw-trie 26MB heap	1.25	2.24	2.62	0.73	1.36
lzw-hash	1.20	2.60	2.63	0.70	1.42
lzw-it1	17.28	31.35	42.28	10.11	0.09
lzw-it2	2.43	4.56	5.64	1.42	0.70
hscopy	0.45	0.88	0.85	0.26	3.80
ccopy	0.05	0.08	0.08	—	—
pure-c	0.11	0.22	0.30	.	.
pure-hs	0.75	1.72	1.78	.	.

[a]The program **lzwc** does not terminate on input "geo".

Figure 1: Runtimes in seconds.

with **hscopy** we have measured a Haskell program which only copies its input to its output and **ccopy** is the corresponding C program (see below for more information). The line **pure-c** contains the difference of **compress** and **ccopy**, the line **pure-hs** contains the difference of **lzw-hash** and **hscopy**. The I/O corrected slowdown factor of Haskell wrt. C is therefore between 6 and 7.8.

The particularly bad performance of **lzw-it1** is due to the inefficient implementation of mutable variables in `ghc-0.23`. The datatype employed in that encoding function is just the naïve imperative transcription of Sanders' datatype:

```
data IPT s a b =
        IPTempty |
        IPT (IPTnode s a b) (IPTree s a b) (IPTree s a b)

data IPTnode s a b =
        IPTnode a b (IPTree s a b)

type IPTree s a b = MutableVar s (IPT s a b)
```

Each mutable variable is implemented as a mutable array with exactly one element. Thus, instead of simply using one additional reference cell, it needs memory for the (trivial) bounds of the array, too. Therefore, allocating and initializing a single mutable variable mean allocating and initializing an mutable array with (at least) three elements. Furthermore, each access to the variable is subjected to unnecessary bounds checks. In **lzw-it2** we have merely collected the three mutable variables into one mutable array, flattened (= removed) the datatype `IPTnode`, and hardcoded the first character by using a mutable array of `IPT2 s a b` as root table:

```
data IPT2 s a b =
        IPT2empty |
        IPT2node a b (IPTree2 s a b)

type IPTree2 s a b = MutArr s Int (IPT2 s a b)
```

Allocating a new node of the tree look like this

```
freshNode2ST k v tree idx =
        newArr (0,2) IPT2empty 'thenST' \ vec ->
        writeArr tree idx (IPT2node k v vec)
```

as compared to the version using IPTnode:

```
freshNodeST k v tree =
        newVar IPTempty 'thenST' \ v1 ->
        newVar IPTempty 'thenST' \ v2 ->
        newVar IPTempty 'thenST' \ v3 ->
        writeVar tree (IPT (IPTnode k v v1) v2 v3)
```

Notice that small changes in coding eg the program lzw-trie can affect runtime and memory usage by a factor of up to 4. To obtain good results intimate knowledge of the properties and inner workings of the transformers is essential. You cannot willy-nilly throw in some state transformers and expect your program to run fast.

As an example consider the function thenST, which performs sequential composition of two transformers, ie it puts one transformer "behind" another. The point to notice with this composition is that it is lazy: in an expression `m 'thenST' f` the function `f` (if it is not strict) can deliver a result even if `m` has not yet started to execute[10]. This happens in the expression `returnST (code: morecode)` in function enter. Here, a code word is available as soon as it is generated; only further demand on the resulting list forces the computation of morecode.

However, the delayed composition thenST is not really necessary for this program to run correctly. If we replace thenST by its strict variant thenStrictlyST the programs runs faster by a factor of two. The price is an increase in heap usage by a factor of four and an increase in stack usage (because some functions are now strict) by factor of five (the program is executable in 5MB heap and 1MB stack). The reason is that the complete list of code words is now constructed in function enter before the first code word appears on the output. The remedy is to switch back to the lazy thenST at the place mentioned above: now lzw-trie(paper1) runs in 2.66 seconds (in 8MB heap)—a slowdown by about 30%— and is executable in 5MB heap and 200k stack.

A more dramatic change occurs in the version using the hash table. Using the lazy thenST it runs in just 200k of heap (but it takes 26 sec). With 4MB of heap 3.86 sec are necessary. The number in the table has been measured with 8MB heap, 2MB stack, and strict thenStrictlyST.

In the version with the hash table, the choice of the hash function and the size of the hash table does not influence the runtime significantly. The changes are in the range of 10%. All of the hash functions use 2–3 arithmetical operations, inspired by the implementation of compress.

Using staged arrays does not pay. All the results (time and space usage) get worse. None of the times in Fig. 1 have been measured using staged arrays.

In order to document the asymptotic logarithmic runtime behavior of the program lzwh, we have performed measurements with prefixes of different length of the text "book1" (size 670k). It appears (cf. Fig. 2) that the pure functional version performs surprisingly well in comparison to the imperative implementations using tries and hash tables. All graphs look linear. The data for lzw-trie appears to get corrupted in the last measurement. This effect is probably due to thrashing.

Finally, we should not ignore that the size of the heap has significant influence on the runtime of the programs. For example, lzw-trie on "paper1" can be accelerated to run in 1.2 seconds, although at the price of 26MB heap memory. Albeit interesting such a measurement is not realistic (yet?).

Enlarging the stack space has no influence on the runtime. The use of the strict thenStrictlyST is the direct cause for the increase in stack space usage in our examples.

[10] If `f` tries to execute an operation on mutable objects complete execution of `m` is forced.

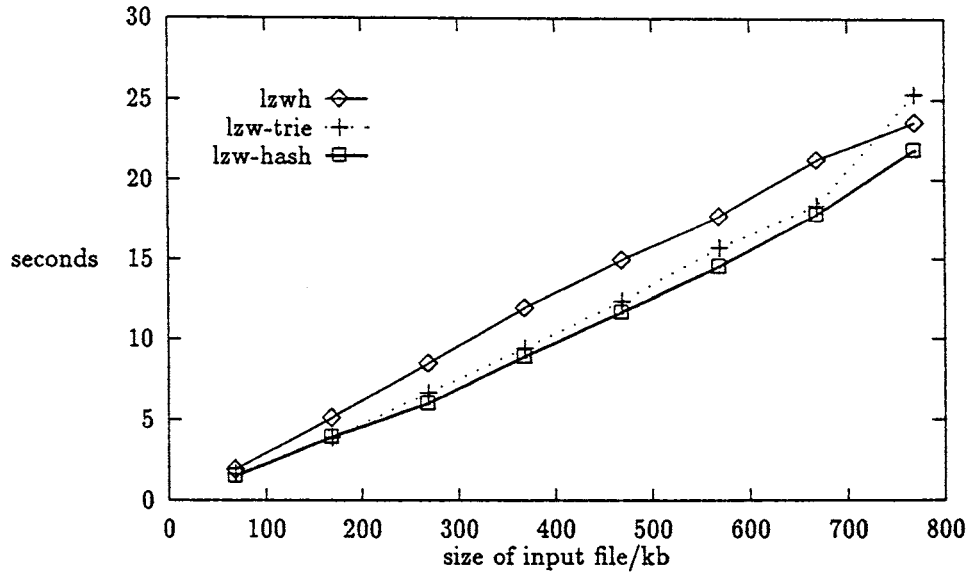

Figure 2: Runtimes for prefixes of book1.

7 Impact on Programming Style

It is quite hard to measure the change in programming style in the presence of laziliy mutable data. In numbers, the core of Sanders' purely functional implementation from [13] consists of 26 lines of code, while the core of ours (the declaration of `Table` and `VecTable` and functions `enter` and `encode`) consists of 22 lines of code.

In the eyes the author, the program in the current text does not use any contorted constructions. The only construction which could be more natural is the test on the contents of a variable, eg:

```
readVar tree `thenST` \ label ->
case label of ...
```

Here, SML's syntax for lambda expressions would be more appropriate.

8 Haskell vs. C

In this section we briefly compare the performance of Haskell against the performance of C. This section is somewhat outside of the scope of the paper which examines the benefits of using imperative extensions provided in some dialects of Haskell. It is only included for completeness and for purposes of comparison with Sanders' work[11].

8.1 I/O

We have written two one-liner programs copy—a C program which copies standard input to standard output using `getchar` and `putchar`—and hscopy—a Haskell program which does the same using `readChan` and `appenChan`. Here is the result, again using `gcc` and `ghc` with optimization.

program	throughput (bytes/sec)	factor
copy	1,830,407	14.28
hscopy	128,128	

[11] Besides my own curiousity, there is another reason which unfortunately escapes my mind temporarily.

Here is our prime suspect for the inefficiency of hscopy: the functions `readChan` and `readFile` both "return" a value of type `[Char]`, ie a lazy list of lazy characters. The list itself should of course be lazy, but the characters should not be lazy! Why? Whenever a character is (physically) read by the system it is immediately available. However, the elements of `[Char]` may not be available and must at least be boxed. Therefore, the input function must box every character, and every function that uses such a character must evaluate/unbox it again. What a useless effort! We suspect the boxing and unboxing operations to account for most of the slowdown experienced in the comparison of copy and hscopy.

A nice solution would be to provide input functions which return lists of unboxed (or strict) characters (ie of type `[Char#]` in GHC terminology), however, as unboxed data types are currently integrated in Glasgow-Haskell polymorphic types cannot be instantiated with unboxed types [10].

8.2 Bit Operations

It was mentioned before that the lack of bitwise operation is one possible source of inefficiencies in the Haskell program, compared to the C program. Bit operations are used in the C program to change the bitstream output of the algorithm into an octet-stream. In the present case we only have to convert a stream of 12bit integers into an octet-stream (see function `recode`, Sec. A).

In order to pinpoint the effect of the conversion on runtimes, we have conducted the following experiment: in the fastest program lzw-hash we have substituted the function `recode` once by a function `crunch` which eagerly devours its input list and returns a constant string, and also by a function `crunch1` which devours its input list and returns a list of the same length as `recode` would have returned. The program lzw-hash* uses the former, while lzw-hash# uses the latter to generate output.

```
crunch (n:ns) | n == n     = crunch ns
              | otherwise = "False"
crunch [] = "True"

crunch1 (n:ns) = 'X': crunch2 ns
crunch1 []     = ""
crunch2 (n:ns) = 'Y':'Z':crunch1 ns
crunch2 []     = "Y"
```

The difference between lzw-hash and lzw-hash* is the time needed for the format conversion and output. The difference between lzw-hash and lzw-hash# is the time needed for the bit operations. The same inputs are used as before[12]. Differences and ratios are computed used the "bib" column.

program \ input	paper1	bib	geo	lzw-hash−x	x/lzw-hash
lzw-hash	1.30	2.80	2.95	0.00	1.00
lzw-hash*	1.10	2.36	2.19	0.44	0.82
lzw-hash#	1.20	2.55	2.49	0.25	0.88

It can be seen from the table that the division and modulo operations from `recode` account for about 12% of the total runtime. Furthermore, output appears to take 6–10%.

9 Conclusion

A number of conclusions can be drawn from this case study.

1. The purely-functional algorithm lzwh compares surprisingly well with the variants programmed in the imperative style. Its runtime is only about 30% slower than the runtime of the fastest (imperative) Haskell-program. This is not much in a time where your new computer runs 2–4 times faster than your old one.

[12] The times for lzw-hash are not identical to the ones shown before, the measurements were done on a different machine.

2. The fastest among the Haskell programs is the implementation using the (mutable) hash table. It cannot be beaten even under unrealistic circumstances (26MB heap).

3. However, lzw-hash is slower than a simple C-program, by a factor of 6.67.

   This factor is still significant, but in the work of Sanders [13] three years ago the factor was still 20–30. Convincing evidence for the rapid advance in compiler technology for functional languages.

4. Imperative algorithms can be formulated in Haskell in a clear, compact, and modular way. This modularity (the algorithm itself, recoding the list of code words into a list of characters, and I/O are separate parts of the program) is mainly achieved through lazy evaluation. The price to pay is some decrease in efficiency.

   Due to the latter fact, experiments with data structures (delayed arrays and staged arrays in our case) can be performed in a short amount of time. Such experiments can be quite time consuming in other languages.

5. Imperative extensions of Haskell should only be employed, if it really cannot be avoided. Making use of them in a correct and efficient way requires some care, since the operations have subtle properties.

6. There are applications where imperative functional programming facilitates the implementation of algorithms, which could not be realized before in pure functional languages (or not in bearable runtime). These applications are in the areas of user interfaces, programming interaction, and interfacing the operating system.

As an extension of the experiments reported here we could imagine the following points:

- An investigation of the space usage of all the programs is necessary.
- Implementation of one of the imperative (trie or hash table) algorithms where I/O is intermingled (as in the C implementation) with the algorithm itself. In the current Haskell 1.3 I/O proposal all I/O-actions are caused through transformers on a special state (the "real world"). All of these transformers are strict, since there is no point in delaying I/O-actions.

  The resulting program is expected to be faster than the Haskell-programs discussed above and it will be more economic in terms of space. However, its structure will be very similar to that of a conventional C-program, and it must be asked where the advantage of using a functional language remains.

Thanks to Will Partain, who provided the programs from [13], and Michael Sperber for inspiring discussions on the LZW-algorithm and on its implementation in Haskell. He still believes that Icon provides for the most elegant implementation of LZW. Last not least, thanks to the referees who did a good job in a short amount of time.

References

[1] Timothy C. Bell, Ian H. Witten, and J.G. Cleary. Modeling for text compression. *ACM Computing Surveys*, 21(4):557–591, December 1989.

[2] The Calgary Text Compression Corpus. Available by ftp on ftp.cpsc.ucalgary.ca in /pub/projects/text.compression.corpus/text.compression.corpus.tar.Z.

[3] Report on the programming language Haskell, a non-strict, purely functional language, version 1.2. *SIGPLAN Notices*, 27(5):R1–R164, May 1992.

[4] John Hughes. Why functional programming matters. *Computer Journal*, 32(2):98–107, 1989.

[5] David J. King and John Launchbury. Structuring depth-first search algorithms in haskell. In *Proc. 22nd ACM Symposium on Principles of Programming Languages*, San Francisco, CA, January 1995. ACM Press.

[6] John Launchbury and Simon L. Peyton Jones. Lazy functional state threads. In *Proc. of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 24–35, Orlando, Fla, USA, June 1994. ACM Press. ACM SIPLAN Notices, v29, 6.

[7] John Launchbury and Simon L. Peyton Jones. State in haskell. *Lisp and Symbolic Computation*, 1995. to appear.

[8] Will Partain. Compression programs, February 1994. electronic mail.

[9] Simon L Peyton Jones, Cordelia Hall, Kevin Hammond, Will Partain, and Philip Wadler. The Glasgow Haskell compiler: a technical overview. In *Proceedings of the UK Joint Framework for Information Technology (JFIT) Technical Conference*, Keele, 1993.

[10] Simon L. Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. In John Hughes, editor, *Proc. Functional Programming Languages and Computer Architecture 1991*, pages 636–666, Cambridge, MA, 1991. Springer-Verlag. LNCS 523.

[11] Simon L. Peyton Jones and Philip L. Wadler. Imperative functional programming. In *Proc. 20th ACM Symposium on Principles of Programming Languages*, pages 71–84, Charleston, South Carolina, January 1993. ACM Press.

[12] Carl Ponder, Patrick McGeer, and Antony Ng. Are applicative languages inefficient? *SIGPLAN Notices*, 23(6):x, 1988.

[13] Paul Sanders and Colin Runciman. LZW text compression in Haskell. In John Launchbury and Patrick M. Sansom, editors, *Proc. of the 1992 Glasgow Workshop on Functional Programming*, pages 215–226, Ayr, Scotland, August 1992. Springer-Verlag, Berlin.

[14] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[15] Peter Thiemann. Safe sequencing of assignments in purely functional programming languages. Technical Report WSI-93-16, Wilhelm-Schickard-Institut, Tübingen, Germany, November 1993.

[16] Peter Thiemann. Terminated references and automatic parallelization for state transformers. In Uday S. Reddy, editor, *ACM SIGPLAN Workshop on State in Programming Languages*, San Francisco, CA, January 1995. University of Illinois.

[17] Terry A. Welch. A technique for high-performance data compression. *IEEE Computer*, 17(6):8–19, June 1984.

[18] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, IT-23(3):337–343, May 1977.

A Changing the Format

Output is produced in pieces of 12 bit each. Other implementations start with pieces of 9 bit and increase the length (on demand) up to 12 bit. Furthermore it is checked (eg by `compress`) whether the rate of compression decreases. If it does so significantly, the hitherto constructed table is dismissed and a new table is constructed, starting out from the empty table again.

The following function `recode` transforms a list of 12 bit `Code` words in a list of 8 bit `Byte`.

```
> recode (n:ns) = chr (n `mod` 256): recode1 (n `div` 256) ns
> recode []     = ""

> recode1 b (n:ns) = chr (b + 16 * (n `mod` 16)): chr (n `div` 16): recode ns
> recode1 b []     = [chr b]
```

... and vice verse:

```
> ercode (b1:b2:bs) = (ord b1 + 256 * (ord b2 `mod` 16)): ercode1 (ord b2 `div` 16) bs
> ercode []         = []

> ercode1 i (b2:bs) = (i + 16 * ord b2): ercode bs
> ercode1 i []      = []
```

B Main Program

The main function, processing the parameters, input and output redirection.

```
> main      = getArgs exit checkArgs

> checkArgs ("-d": rest) = checkFile doDecode rest
> checkArgs ("-e": rest) = checkFile doEncode rest
> checkArgs _ = getProgName
>       (\_    -> appendChan stdout (usage "checkArgs") exit done)
>       (\ str -> appendChan stdout (usage str) exit done)

> usage name = "Usage: " ++ name ++ " (-d|-e) [inputFile [outputFile]]\n"

> checkFile :: (String -> String) -> [String] -> Dialogue
> checkFile proc []             = readChan stdin  exit (checkOutputFile proc [])
> checkFile proc (inFile: rest) = readFile inFile exit (checkOutputFile proc rest)

> checkOutputFile proc [] str = appendChan stdout (proc str) exit done
> checkOutputFile proc (outFile: rest) str = writeFile outFile (proc str) exit done

> doDecode codestring = runST (decode codestring) ""

> doEncode bytestring = recode (runST (encode bytestring))
```

C Representation of the Coding/Decoding Arrays

```
> newEncodeArray       :: (Int,Int) -> b -> ST s (EncodeArray s b)
> newEncodeArrayLazily :: (Int,Int) -> b -> ST s (EncodeArray s b)
> readEncodeArray      :: EncodeArray s b -> Int -> ST s b
> writeEncodeArray     :: EncodeArray s b -> Int -> b -> ST s ()

> type EncodeArray s b = MutArr s Int b
> newEncodeArray       = newArray
> newEncodeArrayLazily = newArrLazily
> readEncodeArray      = readArray
> writeEncodeArray     = writeArray

> newDecodeArray   :: Int -> b -> ST s (DecodeArray s b)
> readDecodeArray  :: DecodeArray s b -> Int -> ST s b
> writeDecodeArray :: DecodeArray s b -> Int -> b -> ST s ()

> type DecodeArray s b = MutArr s Int b
> newDecodeArray hi    = newArray (0, hi)
> readDecodeArray      = readArray
> writeDecodeArray     = writeArray
```

Writing Monads which Manipulate State

Jan-Willem Maessen*

Abstract

More and more, Haskell compilers are offering some form of monadic state manipulation. However, the interfaces to such stateful features as I-structures and M-structures is hardly standardized, and smaller systems (such as Gofer) which are convenient to use for development do not offer imperative features at all. This paper discusses one approach to developing code for two quite different Haskell systems. The first is MacGofer, the well-known Haskell-like interpreter for the Macintosh. The second is parallel Haskell (pH), an eagerly-evaluated, implicitly-parallel dialect of Haskell under development at MIT. The techniques used promise to be useful to others; they also provide a case study in the usefulness of certain extensions to the current Haskell language.

1 Introduction

In recent years, the monadic programming style has taken hold among Haskell programmers. Originally envisioned as a mathematical encapsulation mechanism for semantics of computations [8], monads turn out to be very useful for structuring the computations themselves, particularly when they involve the manipulation of state. I/O [3, 10] and mutable storage [6] can both be implemented safely in Haskell compilers with the appropriate libraries and extensions to the type system.

The intention of this paper is not to propose any radical new monadic extensions to Haskell. Instead, it will examine how the Haskell programmer can make use of various language mechanisms (some of them nonstandard, but none of them new) to implement data structures in a monadic style. This paper uses the example of function memoization, where an obvious fully-functional implementation is much less efficient than one making use of a mutable data structure. In developing the system, it proved useful to break it into subproblems, and solve each using a monad. The original goal of the endeavor was to produce code which would work on in two very different implementations of Haskell-like languages. The final target, parallel Haskell (pH), is an implicitly parallel, eagerly-evaluated dialect of Haskell[9]. However, code was being developed and run on a Macintosh using MacGofer[4], which is sequentially evaluated and contains no imperative extensions at all (and uses the older stream-based I/O model). Writing code which runs cleanly on such different platforms stretches the limits of Haskell's type system, and as such offers an interesting case study in what Haskell (and the Haskell programmer) can and ought to do. Thus, the ultimate goal of this paper is not merely to discuss a particular style of Haskell coding, but also encourage adoption of existing language extensions which will support this style.

*MIT Computation Structures Group, jmaessen@mit.edu. The author is supported in part by an NSF Graduate Fellowship.

```
class Monad m where
    (>>=)  :: m a -> (a -> m b) -> m b
    return :: a -> m a
```

Figure 1: The monadic operations

2 Preliminaries

A few notes about pH will be neccessary before jumping into the paper proper. pH is eagerly evaluated. Roughly speaking, this means that every expression will be evaluated (as in a call by value language), but that non-strict dependencies in the program code are allowed. The practical upshot is that we must *explicitly create thunks*—if we don't wish a particular object to be evaluated, we must turn it into a function. Thus, the following two snippets of code have different meanings in pH:

```
f = (\_ -> hairyComputation)
g = let value = hairyComputation in (\_ -> value)
```

Here `g` will *always* perform `hairyComputation`, regardless of whether `g` is ever applied; `f`, on the other hand, will perform `hairyComputation` every single time it is applied. The explicit creation of thunks will be most evident when we look at types; often we will write `() -> Type` where most Haskell programmers would simply use `Type`.

Since this paper will be talking extensively about monads, it is seems fitting to begin with a quick definition of monadic operations. A certain familiarity with monadic code has been assumed in this paper; this presentation therefore focuses on the notation used, which is summarized in figure 1. In general, we can read the monadic type `m a` as

```
transClosure dag = transNodes dag
  where
    transNode :: DAGNode a -> [DAGNode a]
    transNode (DAGNode a children) =
        DAGNode a newchildren : newchildren
      where newchildren = transNodes children
    transNodes :: DAG a -> [DAGNode a]
    transNodes dag = foldl unionList []
                         (map transNode dag)
```

Figure 2: A naive implementation of transitive closure

"a computation in the monad `m` returning a value of type a." A monad has two fundamental operations. The expression `return v` yields a computation returning value `v`. The operator (`>>=`) obtains the result from its left-hand computation and passes it as an argument to the function on the right-hand side, which in turn returns another computation. The two computations are combined in some manner, constrained by the following identities:

```
unit a >>= f = f a
x >>= (\a -> f a >>= g) = (x >>= \a -> f a) >>= g
x >>= return = x
```

The astute reader will have noticed the use of type constructor classes in the definition of Monads[5]. In the systems being used we can get away with this, since both Gofer and pH provide constructor classes. For the moment, they will simply be a notational convenience; the same monadic notation is used for all the monads defined. This helps emphasize their similarities. Later, constructor classes will be put to heavier use in some "what-if" thought experiments.

Finally, the monad types which are used in this presentation are often unwieldy, containing as many as four free type variables at a time. To help keep things somewhat more clear, the following notation is used:

`s` represents the type of the "state" of an underlying state transformer monad (more on this later).

`v` represents the type of the value returned when the monad's computation is performed, if this type is distinct from the others.

`a`, `b`, `c` represent other type variables, whose meaning is given by context.

3 The problem

Let us attempt to solve a fairly simple problem: Given a directed acyclic graph, define a function `transClosure` which computes the transitive closure of its input (the list of nodes reachable from the root set of the given graph):

```
transClosure :: Hashable a => DAG a -> [DAGNode a]
```

A `DAGNode` is some data (with type `a`) and a list of nodes to which the given node is connected.

```
data Eq a => DAGNode a = DAGNode a (DAG a)
type Eq a => DAG a = [DAGNode a]
```

We can define a notion of equality among nodes, assuming that every `DAGNode` has a unique set of data (perhaps the data includes a distinguishing tag, for example):

```
instance Eq a => Eq (DAGNode a) where
    (DAGNode a _) == (DAGNode b _) = a == b
```

Also defined is a notion of hashing a graph node; the function hash takes a hashable object and returns an integer:

```
instance Hashable a => Hashable (DAGNode a) where
    hash (DAGNode a _) = hash a
```

We make use of hashability in our implementation to keep track of visited nodes.

The most obvious implementation of `transClosure`, shown in figure 2, is poor at best: This version of `countNodes` completely ignores any sharing. For example, consider the following:

```
a = DAGNode aData [b,c]   --   a          a
                          --  / \        / \
b = DAGNode bData [c]     -- |   b  => c   b
                          --  \ /        /
c = DAGNode bData []      --   c          c
```

```
aTrans = transClosure [a]
```

As written, aTrans will compute `transClosure c` twice, leading to loss of sharing in the resulting graph (not to mention lost performance through repeated computation). We would like to provide some means of memoizing the results of `transNode` so that each node in the graph is closed over exactly once.

One way of memoizing calls to `transNode` would be to introduce an explicit data structure which is threaded through all of the recursive calls to `transClosure`. There are a number of objections to doing so directly, however. First, by threading a data structure through the computation it is likely that the entire graph structure will need to be evaluated before any part of the final answer can be examined. Just as important, the eventual goal is to produce a *parallel* implementation of the algorithm; with computation forced to wait for the threaded data structure, not much can be going on at any one time. Finally, the mechanics of *how* the data structure is threaded through the computation are likely to change if the data structure is changed, requiring extensive modifications to the structure of the algorithm itself.

4 A memoization monad

A monadic implementation addresses all three of these concerns by hiding the necessary data structures and the manner in which those data structures are accessed. We can use the code in figure 3. This code assumes the existence of a memoization type of the following form:

```
MemoizingComp s a b v
```

Intuitively, this memoizes a function of type `a->b`, and makes use of some internal state `s`.

The code as written implies several things. First, a given instance of the memoization monad only memoizes calls to a single function. Why? Purely functional implementations of Haskell have no notion of mutable state. This means that such state must be represented explicitly within the monad. As a result, only a bounded number of types could be memoized directly in such an implementation. This apparent expressive difficulty will be explored later.

Second, an instance of the memoization monad is given the function memoized only once, in the following code:

```
type ClosureComp s a =
    MemoizingComp s (DAGNode a) [DAGNode a] [DAGNode a]
transClosure dag =
    memoizing canonicalSize
        {function = transNode;
         computation = transNodes dag}
  where transNode :: DAGNode a -> ClosureComp s a
        transNode (DAGNode a children) =
            transNodes children >>= \newchildren ->
            return (DAGNode a newchildren : newchildren)
        transNodes :: [DAGNode a] -> ClosureComp s a
        transNodes nodes = foldl combineNodes (return [])
                                (map memo nodes)
        combineNodes :: ClosureComp s a ->
                        ClosureComp s a ->
                        ClosureComp s a
        combineNodes nodes1 nodes2 =
            nodes1 >>= \nodes1close ->
            nodes2 >>= \nodes2close ->
            return (unionList nodes1close nodes2close)
```

Figure 3: A memoizing implementation of transitive closure

```
memoizing canonicalSize
    {function = transNode;
     computation = transNodes dag}
```

While at first glance this may look similar to the previous decision, it is actually entirely orthogonal; we can imagine allowing numerous functions to be memoized, but still associating a function specifically with its tabulated results. By suppliying the function exactly once, we guarantee that the results of a "memoized call" actually correspond to the results of the function we wanted to compute.

It is worth emphasizing here that *a* `memoizingComp` *is not itself a memoized function.* Instead, it represents a piece of computation for which we want to maintain memoization information for a particular function. *The actual memoized function is invoked by a call to* `memo`.

We thus define the a type for memoizing computations as follows:

```
> data (Hashable a) => MemoizingComp s a b v =
>     MC ((a -> MemoizingComp s a b b) ->
>         HashTableComp s a b v)
```

For our memoization computation, we really just want to maintain a data structure which will associate each argument of type a with a result of type b. We represent this as a `HashTableComp`, and defer its precise description until later. We want to pass along the (fixed) function being memoized as we perform computation.

Note the rather odd type for the function being memoized:

```
a -> MemoizingComp s a b b
```

Instead of returning a simple value, we assume that the function memoized itself returns a computation! Note that the function we wish to memoize, `transNode`, is recursive. In order to memoize these recursive calls, we need to perform them as memoizing computations. Since we want the results of such memoized calls to be propagated everywhere, `transNode` returns a computation.

This points up a general principle of programming with monads:

Function arguments should return computations.

While in general we may not need this power, it is quite easy to wrap a call to return around a function's value, whereas it is completely impossible to insert a computation where it was not expected. Following this principle makes recursion among computations much easier and more natural to express.

Memoizing computations simply propagate the function being memoized to sub-computations, and "plumb together" the hash table computations which result:

```
> instance Monad MemoizingComp s a b where
>     (MC m1) >>= fm2 =
>         MC (\f -> (m1 f) >>= \r1 ->
>                   let MC m2 = fm2 r1 in m2 f)
```

If a value is simply being returned, we ignore the function being memoized and simply return a hash table computation yielding the appropriate value:

```
>     return v = MC (const (return v))
```

We still need to define how to call the memoized function:

```
> memo :: Hashable a => a -> MemoizingComp s a b b
```

That is, whenever we want to call the function being memoized, we simply write `memo` instead of the function's original name. The implementation of `memo` assumes the existence of a single hash table operation, `lookupInsertHash`:

```
> memo argument =
>   MC (\f -> lookupInsertHash argument
```

When the lookup fails to find a memoized result, we must provide a thunk to compute an appropriate value; `lookupInsertHash` will then memoize and return it:

```
>               (\_->
>                 let (MC fHashComp) = f argument
>                 in  fHashComp f))
```

Here, the memoizing computation `(MC fHashComp)` is explicitly converted into a `HashTableComp` by applying it to `f`. In effect, we are taking a function, `f`, returning a value in one monad (`MemoizingComp s a b`) and using it to create a function (the thunk) returning a value in another monad (`HashTableComp s a b`). This *must* be part of the implementation of memoizing computations, since it requires knowledge of the internal structure of `MemoizingComp`.

The `loookupInsertHash` function looks its first argument up in the hash table, and returns the value of the resulting entry if one exists. If there is no entry, one is created containing the value of the thunk passed as an argument. Here we see a case where pH differs from Haskell. In Haskell, we could simply provide the value directly, and rely on laziness to gaurantee that this value is not computed unless it is necessary. Eager evaluation forces us to explicitly build thunks to do so. Thus, we know that `lookupInsertHash` has the following type:

```
lookupInsertHash :: Hashable a =>
    a -> (() -> HashTableComp s a b b)
      -> HashTableComp s a b b
```

One function remains to be defined, and this is where the problems really occur. We would like to design a function something like:

```
remembering function (MC computation) =
      compute (computation function)
      run the resulting HashTableComp to obtain an answer.
```

The problem is that our eventual implementation is going to want to manipulate mutable state using a state transformer. In order to evaluated such a computation safely, a function of the following form must be used[6]:

```
runST :: ∀v. (∀s.ST s v) -> v
```

State transformer computations need to be universally quantified over their state type, s. In order to ensure this quantification, we are forced to bring s in to all computations which make use of state, and to enforce the same quantification rules for them as well.

Unfortunately, Haskell itself does not provide any support for local quantification of type variables. Rather than reject a "stateful" approach out of hand, or even settle for unsafe features, we make use of a proposed record extension which provides exactly the facilities we need. We will create a "wrapper" record which expresses the type relationships for memoizing computations:

```
> data Hashable a => MemoWrapper a b v =
>      { function :: a -> MemoizingComp s a b b;
>        memoComputation :: MemoizingComp s a b v }
```

Here the type of the state, s, is not a parameter of the wrapper type. The rules for the record system state that such type variables must be universally quantified; records are typechecked as if they were let blocks. Two accessor functions, function and memoComputation, are defined by the compiler. Thus, a new language feature—named records—enables us to solve an existing problem that would otherwise have required special modifications to the type checker.[1] The function memoizing actually evaluates a wrapped-up computation:

```
> memoizing :: Hashable a =>
>      Int -> MemoWrapper a b v -> v
> memoizing size wrapper =
>      withHash size { hashComputation = comp f }
>    where (MC comp) = memoComputation wrapper
>          f         = function wrapper
```

Note that a similar argument applies to HashTableComps as well, so a similar wrapper record has been defined for them.

Having shown how useful record types are for describing wrappers such as memoizing, it is worth discussing their downside—that they remain unimplemented on one of the two systems (Gofer) for which we are trying to write code![1] This means that the relevant code will need to be rewritten—both memoizing and transClosure. Luckily, the striking (and purposeful) similarity between records and let blocks comes to our rescue:

```
transClosure dag =
    memoizing canonicalSize
        let {function = transNode;
             computation = transNodes dag}
        in (function, computation))
  where ... -- as before
```

This is, roughly speaking, the translation which would be performed by the type checker on record types anyway; we've simply lost the local quantification of state. However, we are not using state transformers in this particular system, and thus this does not hurt us in practice. We can rewrite memoizing as follows (the original structure is preserved for comparison):

```
data MemoWrapper s a b v =                          -- *
    ( a -> MemoizingComp s a b b,
      MemoizingComp s a b v )
memoizing :: Hashable a =>
    Int -> MemoWrapper s a b v -> v                 -- *
memoizing size wrapper =
    withHash size
            let { hashComputation = comp f }
            in  hashComputation                     -- *
  where (MC comp) = memoComputation wrapper
        f         = function wrapper
        function  = fst                             -- *
        memoComputation = snd                       -- *
```

Nevertheless, while the rewriting needed is simple, it would be nicer if it were nonexistent. This will require one of two changes: either introduce local universal quantification of types separately into Haskell (this could become very cumbersome indeed!), or implement more widespread support for a record system such as this which includes such types for free. Regardless of which course is taken, it is absolutely clear that support for local quantification is needed to make state-manipulating programs run safely.

5 Hash Tables

The implementation of memoization was very brief—if the code were consolidated, it would fit on half a sheet of paper. Moreover, the code for both Gofer and pH was similar—modulo the need for local quantification in the pH code. This is hardly surprising, however, since the MemoizingComp monad does very little actual work. The difficult job is actually implementing an updatable hash table in a monad.

In Gofer, we make use of an IntMap to store the hash table, updating the structure incrementally each time a new entry is created (figure 4). Note that the current value of the hash table is threaded through the computations in >>= from left to right. We could just as easily have threaded it through in the opposite direction instead, giving the following alternative definition of >>=:

```
(HT h1) >>= fht =
    HT \table->
        let (table1, h1res) = h1 table2
            (HT h2) = fht h1res
            (table2, h2res) = h2 table
        in  (table1, h2res))
```

In fact, when testing higher-level code which will later be run in parallel, it is useful to try both definitions of >>=—the result returned by the program should be exactly the same in either case. Indeed, it is possible (though tedious) to write a version of the hashTableComp monad which tries every possible permutation of the orders of >>= operations

[1] Ironically, Lennart Augustsson credits Mark Jones for the record system upon which the one in pH is based—it simply does not appear in any of the versions of Gofer I used.

```
module HashTable(withHash, getHash, lookupInsertHash, stToHash, HashTableComp) where
import Monad

type Hashable a => HashTable a b = IntMap [(a,b)]

data Hashable a => HashTableComp s a b v =
        HT (HashTable a b -> (HashTable a b, v))

instance Monad (HashTableComp s a b) where
  (HT h1) >>= fht = HT (\table->
                            let (table', h1res) = h1 table
                                (HT h2) = fht h1res
                            in  h2 table')
  return v = HT (\t -> (t,v))


withHash :: (Hashable a) => Int -> HashTableComp s a b v -> v
withHash size hc =
    snd (comp emptyMap)
  where HC comp = hc


lookupInsertHash ::
    (Hashable a) => a -> (() -> HashTableComp s a b b) -> HashTableComp s a b b
lookupInsertHash element valueThunk =
    HT \table ->
            case lookupMap table elementHash of
                Nothing -> newEntry insertMap [] table
                Just entries -> case lookupAssoc entries element of
                    Nothing -> newEntry replaceMap entries table
                    Just value -> (table, value)
  where elementHash = hash element
        newEntry modification entries table = ret
          where HT valueComp = valueThunk()
                ret@(table2, value) = valueComp table1
                table1 = modification table elementHash [(element, value)]
```

Figure 4: An implementation of the HashTableComp monad in Gofer

in the program. This could be used to exhaustively verify correctness and determinacy if desired.

Many of the notes and caveats which have been discussed for MemoizingComps also apply to HashTableComps. We have already mentioned that different versions of the withHash function are neccessary under Gofer and pH; similarly, the use of a thunk returning a computation as an argument to lookupInsertHash follows the practices we've seen so far. Once again, for the sake of type uniformity in the Gofer version, only a single hash table is actually represented by the monad. And once again, the function lookupInsertHash takes a function returning a computation rather than a value.

The pH implementation simply uses an MVector to represent the hash table (figure 5). An MVector is a reference to a chunk of mutable storage which is indexed from 0.[2] MVectors can only be manipulated by state transformer operations. It is for this reason, and this reason alone, that we needed to introduce the type of the state, s, into all of our monads thus far:

```
data ST s a
data MVector s a

instance Monad (ST s)

data Wrapper a = {computation :: ST s a}
statefully :: Wrapper a -> a
```

Here, the function runST discussed above and in [6] has been replaced by the function statefully, which takes a Wrapper record and performs the computation it contains. In this way, a state transformer computation can be run safely without further extending the type system beyond adding record types.

Two functions are used to create and manipulate MVectors. The expression makeMVector size [], in withHash, returns a computation which creates a new MVector of size size (meaning its maximum index is size-1), all of whose elements are initialized to []. Thus makeMVector has the following type:

```
makeMVector :: Int -> a -> ST s (MVector s a)
```

The function mVModify is used to atomically modify the value of an element of an MVector. Its arguments are a vector, an index, and a function. The function has the following type:

```
a -> ST s (a,b)
```

That is, it takes as an argument the current value of the appropriate element of the vector, and yields a computation which computes both a new value for that element and a result. The computation has one contraint: at no time can the same element of the vector be accessed via a nested call to mVModify. That is, the vector element "vanishes" when it is in the process of being modified. This is because there is no particular ordering (beyond data dependency) in which mVModify operations occur. Thus, in order to guarantee that the value seen is the most recent one, we provide an atomic fetch-compute-store operation:

```
mVModify ::
    MVector s a -> Int -> (a -> ST s (a,b)) ->
        ST s b
```

```
class Monad m => FixMonad m where
  fixMonad :: (v -> m v) -> m v

instance FixMonad (HashTableComp s a b) where
  fixMonad f = HT (\size table ->
                     let (HT comp) = f res
                         res = comp size table
                     in  res)

instance FixMonad (MemoizingComp s a b) where
  fixMonad f = MC (\function ->
                     let (MC comp) = f res
                         res = comp function
                     in  res)
```

Figure 6: Monadic fixed points

Sometimes, however, we need to perform some computation which depends *non-strictly* on the result of further calls to mVModify. In the case of lookupInsertHash, each hash table entry is a list of objects and values which hashed to that slot in the table. When an entry for an object does not exist, one is created inside the call to mVModify:

```
let lookup = lookupAssoc entries element
    newvalue =
      case lookup of
        Nothing -> (element, returnedValue):entries
        Just _  -> entries
in return (newvalue, lookup))
```

The actual value is computed *outside* the call, however—the function being memoized can call itself recursively, and if one of the recursive calls causes a collision in the hash table, the recursive call to mVModify will deadlock unless we can insure that the original call to mVModify has completed:

```
case lookup of
  Nothing    -> makevalue size table
  Just value -> return value
```

Getting this to work requires the function fixMonad which feeds the value of a monadic computation back as an input to that computation. Not every monad has a fixed point operation fixMonad; the exact implementation depends on the monad itself. For example, both of the monads described in this paper have a valid definition for fixMonad as shown in figure 6. For lookupInsertHash, the call to fixMonad simply feeds the result of the above expression back to the call to mVModify so that the value can be added to the hash table if necessary.

6 Related Work

The problem which provided the inspiration for the example in this paper was originally an assignment in Arvind's dataflow course. The monadic features of pH are distinctly *ad hoc*, and exist in a seperate library from much of the rest of the language. They are coded using the better-documented I-structures and M-structures of pH itself[9].

[2]Note that MVectors are less general than Haskell arrays in this respect; implementing a type MArray(with general indexing) in terms of MVector, or vice versa, is trivial.

```
module HashTable(withHash, getHash, lookupInsertHash, stToHash, HashTableComp) where
import Monad

data (Hashable a) => HashTableComp s a b v =
        HT (Int -> MVector s [(a,b)] -> ST s v)

instance Monad (HashTableComp s a b) where
  (HT h1) >>= fht = HT (\size table->
                            let (HT h2) = fht (h1 size table)
                            in  h2 size table)
  return v = HT (\_ _ -> v)

data HashWrapper a b v = { hashComputation :: HashTableComp s a b v }

withHash :: (Hashable a) => Int -> HashWrapper a b v -> v
withHash size hc =
    statefully {computation = mVector size [] >>= (\table -> comp size table)}
  where HT comp = hashComputation hc

lookupInsertHash ::
    (Hashable a) => a -> (() -> HashTableComp s a b b) -> HashTableComp s a b b
lookupInsertHash element valueThunk =
    HT \size table ->
        fixMonad \returnedValue ->
            mVModify table (elementHash `mod` size) $ (\entries ->
                let lookup = lookupAssoc entries element
                    newvalue =
                        case lookup of
                            Nothing -> (element, returnedValue):entries
                            Just _  -> entries
                in return (newvalue, lookup))
                                                    >>= \lookup ->
            case lookup of
              Nothing    -> makevalue size table
              Just value -> return value
  where elementHash = hash element
```

Figure 5: An implementation of the HashTableComp monad in pH

Some of the features described in this paper—particularly the record system—only exist in preliminary form at the moment.[1]

Over the past few years, a tremendous number of papers have appeared describing various uses for monads in functional programming. Wadler's paper on monad comprehensions [13] is seminal in this area. A large body of papers describe the use of monadic code to describe impure language features, including I/O[10, 6], mutable storage[6], and data parallelism[2]. The techniques of Launchbury and Peyton Jones[6] have been influential in developing the underlying pH state transformer, especially in using free type variables and local quantification to delimit the validity of mutable references.

While a large informal body of knowledge about various monadic programming techniques (as opposed to language features discussed above) seems to exist, this author knows of very few written examples. Those which do exist and are well known include the work of Wadler[12], Steele[11], and Liang *et. al.*[7] in using monads to structure interpreters. The latter, in which monads are layered on top of one another as just as they are in this work, is most relevant for current purposes. Some important differences exist, of course. Constructor classes are used far more heavily to define the various entities involved. The paper describes layering monads systematically—that is, every monad transformer is parameterized with respect to some underlying monad in terms of which it is implemented, and what matters is not *what monad is used*, but rather *what functionality is provided.* This distinction is important, and is the initial subject of the next section.

7 Challenges and Future Work

What we want out of a memoizing computation is really just the functionality of `memo`. Realizing this, we could simply define a type class which provides this functionality:

```
class MemoizingComp m where
    memo      :: a -> m a b b
```

We can then define the version of memoization discussed here:

```
data (Hashable a, HashTableComp h) =>
     MyMemo h a b v =
    MC ((a->MyMemo h a b b) -> h a b v)

instance HashTableComp h =>
         MemoizingComp (MyMemo h) where
    memo ...
```

Note that this approach gets rid of the extra type variable `s` when it is not needed by any of the underlying monads (definitely a win in Gofer!). The types discussed in this paper might end up being written:

```
type GoferMemoComp a b v =
      MyMemo GoferHashComp a b v
type PHMemoComp s a b v  =
      MyMemo (PHHashComp (ST s)) a b v
```

There's one problem here: we've said nothing about the fact that `MemoizingComp` must be a monad! There's no way to write:

```
class Monad (m a b) =>
      MemoizingComp m where
```

There is undoubtedly an approach which is better than this and which will work to capture the notion that a `MemoizingComp` must instantiate to a monad.

The possibility of requiring a particular *functionality* for a given computation also means that we can write monads which provide *several* functionalities at once. The simplest imaginable way of doing so is to open up certain aspects of the underlying monads so that they are visible in the monad actually being used. For example, we can imagine providing "hooks" in any monad using a state transformer, so that the underlying state transformer can also be accessed. This might look something like the following:

```
instance StateTransformer st,
         HashTableComp h =>
         StateTransformer h (st s) where ...
```

Some readers have probably noticed with alarm the complexity of the types involved in this paper. Each of our monads has no less than three free type variables! This seems to be a problem in general with monadic code when it is written in a general style. When we use a monad, we will want to instantiate its type to the type at which we actually plan on using it; for example, this declaration was found at the beginning of the memoized version of `transClosure`:

```
type ClosureComp s a =
    MemoizingComp s (DAGNode a) [DAGNode a] [DAGNode a]
```

Of course, we'd like to spend most of our time writing useful computations like `transClosure`, and not designing the infrastructure allowing us to support them. So perhaps we should be willing to concede that huge types are a neccessary evil when we are writing the monadic infrastructure we use. Nonetheless, there are some possible ways to make the techniques less opaque. For example, it might make more sense to parameterize `MemoizingComp` with a function type, as follows:

```
data NewMemoComp s f r =
    MC ((a -> NewMemoComp s (a->b) b) ->
        HashTableComp s a b r)
    :: NewMemoComp s (a->b) r
```

Here the data constructor `MC` has been given an explicit return value; thus, there will be no way to meaningfully construct a `MemoComp` which *isn't* parameterized by a function type.

Even more worrisome than the exploding sizes of types is the enormous number of data constructors introduced by monadic code. It would be nice to be able to define a monad like `MemoizingComp` without an explicit constructor:

```
type MemoizingComp s a b v =
    (a -> MemoizingComp s a b v) ->
    HashTableComp s a b b
```

Unfortunately, Haskell does not permit us to give an instance declaration for `Monad MemoizingComp`. The use of abstype declarations in place of the data declarations used in this paper make the types involved semantically equivalent; nonetheless, the syntactic "noise" introduced by the data constructor remains.

Finally, a plea which is perhaps more farfetched. Earlier in this paper, it was noted that the State Transformer monad was special, in that it separated the type of mutable state (associated with the reference to that state, e.g. `MVector`

s a) from the state itself (propagated implicitly throughout state transformer computations). There is no way to do this for an arbitrary type *without* using state transformers! If there were, monads could provide many instances of the state they encapsulate, each instance at a different type. The problem is similar to the one of dynamic typing; however, it is possible that by limiting type information solely to references, such references will be typable at compile time and things will still work nicely. Finding a general solution to the problem will, for example, permit a type-correct implementation of the state transformer monad *entirely within the functional part of Haskell.*

8 Conclusion

This paper has explored some of the issues involved in writing monadic code, with the idea that such code can be made fairly portable between different versions of Haskell. Several pointers for other programmers can be observed in this experience. First, abstraction is as useful with state-manipulating code as it is elsewhere. Of course, this ought to have been obvious beforehand! More useful are the design principles employed. When writing monadic code, it is important to identify features (such as the function being memoized in a `MemoizingComp`) which ought to be fixed, and therefore encapsulated by the monadic computation itself. Higher-order functions used in a monad should themselves return computations in that monad; this permits such functions to perform additional manipulations based on their arguments.

Actually sitting down and trying to write monadic code also leads to some observations about various features of Haskell itself. Type constructor classes already provide a crucial degree of flexibility in Haskell implementations which support them (and most do). Abstract type declarations would make them even more useful and cut down on syntactic "noise" in programs. Finally, record types or some equivalent system for obtaining local quantification of types will be essential in writing extensible monads which manipulate mutable storage.

Finally, this paper has focused on monadic programming, but it should be stressed that monads are not The Answer. Even when written carefully and systematically, monadic programming tends to be tedious and error-prone. While most of these errors are caught by the type checker, the fact remins that monads are probably best avoided in most code. It will doubtless become clearer with the passage of time when and how monadic code ought to be used. It is clear that when used carefully, a monadic coding style can simplify and streamline writing code which manipulates mutable storage.

References

[1] Lennart Augustsson. Record types in the hbcc compiler. Personal communication with the author, April 1995.

[2] Keith Clarke and Jonathan M D Hill. Parallel haskell: The vectorization monad. Look me up.

[3] Kevin Hammond, et. al. Report on the programming language haskell, version 1.3. To be released at FPCA '95, 1995.

[4] Mark Jones. Gofer language manual. No formal reference known.

[5] Mark Jones. Type constructor classes. In *Proceedings of the 6th ACM Conference on Functional Programming Languages and Computer Architecture*, 1993.

[6] John Launchbury and Simon L Peyton Jones. Lazy functional state threads. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 24–35. ACM SIGPLAN Notices, 1994.

[7] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 333–343, 1995.

[8] Eugenio Moggi. Computational lambda-calculus and monads. In *IEEE Symposium on Logic in Computer Science*, pages 14–23, 1989.

[9] Rishiyur S Nikhil, Arvind, and James Hicks, et. al. ph language reference manual, version 1.0—preliminary. Technical Report 369, MIT Computation Structures Group Memo, January 1995. Working document describing pH extensions to Haskell.

[10] Simon L Peyton Jones and Philip Wadler. Imperative functional programming. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 71–84, 1993.

[11] Guy L Steele Jr. Building interpreters by composing monads. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 472–492, 1994.

[12] Philip Wadler. The essence of functional programming. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 1–14, 1992.

[13] Philip Wadler. Comprehending monads. *Mathematical Structure in Computer Science*, To appear.