Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell

Simon PEYTON JONES

Microsoft Research, Cambridge

`simonpj@microsoft.com`

`http://research.microsoft.com/users/simonpj`

April 7, 2010

Abstract

Functional programming may be beautiful, but to write real applications we must grapple with awkward real-world issues: input/output, robustness, concurrency, and interfacing to programs written in other languages.

These lecture notes give an overview of the techniques that have been developed by the Haskell community to address these problems. I introduce various proposed extensions to Haskell along the way, and I offer an operational semantics that explains what these extensions mean.

This tutorial was given at the Marktoberdorf Summer School 2000. It will appears in the book *"Engineering theories of software construction, Marktoberdorf Summer School 2000"*, ed CAR Hoare, M Broy, and R Steinbrueggen, NATO ASI Series, IOS Press, 2001, pp47-96.

This version has a few errors corrected compared with the published version. Change summary:

- Jan 2009: Clarify ν and $fn\ ()$ in Section 3.5; reword the one occurrence of $fv\ ()$ in Section 2.7
- Feb 2008: Fix typo in Section 3.5
- May 2005: Section 6: correct the way in which the FFI declares an imported function to be pure (no "`unsafe`" necessary).
- Apr 2005: Section 5.2.2: some examples added to clarify `evaluate`.
- March 2002: substantial revision

1 Introduction

There are lots of books about functional programming in Haskell [44, 14, 7]. They tend to concentrate on the beautiful core of functional programming: higher order functions, algebraic data types, polymorphic type systems, and so on. These lecture notes are about the bits that usually *aren't* written about. To write programs that are *useful* as well as *beautiful*, the programmer must, in the end, confront the Awkward Squad, a range of un-beautiful but crucial issues, generally concerning interaction with the external world:

- Input and output.
- Error detection and recovery; for example, perhaps the program should time out if something does not happen in time.
- Concurrency, when the program must react in a timely way to independent input sources.

- Interfacing to libraries or components written in some other language.

The call-by-value (or strict) family of functional languages have generally taken a pragmatic approach to these questions, mostly by adopting a similar approach to that taken by imperative languages. You want to print something? No problem; we'll just have a function `printChar` that has the side effect of printing a character. Of course, `printChar` isn't really a function any more (because it has a side effect), but in practice this approach works just fine, provided you are prepared to specify order of evaluation as part of the language design — and that is just what almost all other programming languages do, from FORTRAN and Java to mostly-functional ones like Lisp, and Standard ML.

Call-by-need (or lazy) languages, such as Haskell, wear a hair shirt because their evaluation order is deliberately unspecified. Suppose that we were to extend Haskell by adding side-effecting "functions" such as `printChar`. Now consider this list

```
xs = [printChar 'a', printChar 'b']
```

(The square brackets and commas denote a list in Haskell.) What on earth might this mean? In SML, evaluating this binding would print `'a'` followed by `'b'`. But in Haskell, the calls to `printChar` will only be executed if the elements of the list are evaluated. For example, if the only use of `xs` is in the call `(length xs)`, then nothing at all will be printed, because `length` does not touch the elements of the list.

The bottom line is that *laziness* and *side effects* are, from a practical point of view, incompatible. If you want to use a lazy language, it pretty much has to be a *purely* functional language; if you want to use side effects, you had better use a strict language.

For a long time this situation was rather embarrassing for the lazy community: even the input/output story for purely-functional languages was weak and unconvincing, let alone error recovery, concurrency, etc. Over the last few years, a surprising solution has emerged: the monad. I say "surprising" because anything with as exotic a name as "monad" — derived from category theory, one of the most abstract branches of mathematics — is unlikely to be very useful to red-blooded programmers. But one of the joys of functional programming is the way in which apparently-exotic theory can have a direct and practical application, and the monadic story is a good example. Using monads we have found how to structure programs that perform input/output so that we can, in effect, do imperative programming where that is what we want, and only where we want. Indeed, the `IO` monad is the unifying theme of these notes.

The "standard" version of Haskell is Haskell 98, which comes with an I/O library that uses the monadic approach. However, Haskell 98 is not rich enough to deal with the rest of the Awkward Squad (exceptions, concurrency, etc), so we have extended Haskell 98 in a number of experimental ways, adding support for concurrency [35], exceptions [37, 29], and a foreign-language interface [36, 11]. So far, these developments have mostly been documented in scattered research papers; my purpose in these lectures is to gather some of it together into a coherent account. In what follows, when I refer to "Haskell", I will always mean Haskell 98, rather than earlier versions of the language, unless otherwise specified.

As a motivating example, we will explore the issues involved in writing a web server in Haskell. It makes an interesting case study because it involves every member of the Awkward Squad:

- It is I/O intensive.
- It requires concurrency.
- It requires interaction with pre-existing low-level I/O libraries.
- It requires robustness. Dropped connections must time out; it must be possible to reconfigure the server without dropping running connections; errors must be logged.

The Haskell web server we use as a case study is remarkably small [27]. It uses only 1500 lines of Haskell to implement (more than) the HTTP/1.1 standard. It is robust enough to run continuously for weeks at a time, and its performance is broadly comparable with the widely-used Apache server. Apache handles 950 connections/sec on the machine we used, while the Haskell web server handles 700 connections/sec. But this is a bit of an apples-and-oranges comparison: on the one hand Apache has much more functionality while, on the other, the Haskell web server has had very little performance tuning applied.

I began this introduction by saying that we must confront the Awkward Squad if we are to write useful programs. Does that mean that useful programs are awkward? You must judge for yourself, but I believe that the monadic approach to programming, in which actions are first class values, is itself interesting, beautiful, and modular. In short, Haskell is the world's finest imperative programming language.

2 Input and output

The first member of the Awkward Squad is input/output, and that is what we tackle first.

2.1 The problem

We begin with an apparently fundamental conflict. A purely functional program implements a *function*; it has no side effect. Yet the ultimate purpose of running a program is invariably to cause some side effect: a changed file, some new pixels on the screen, a message sent, or whatever. Indeed it's a bit cheeky to call input/output "awkward" at all. I/O is the raison d'être of every program. — a program that had no observable effect whatsoever (no input, no output) would not be very useful.

Well, if the side effect can't be in the functional program, it will have to be outside it. For example, perhaps the functional program could be a function mapping an input character string to an output string:

```
main :: String -> String
```

Now a "wrapper" program, written in (gasp!) C, can get an input string from somewhere (a specified file, for example, or the standard input), apply the function to it, and store the result string somewhere (another file, or the standard output). Our functional programs must remain pure, so we locate all sinfulness in the "wrapper".

The trouble is that one sin leads to another. What if you want to read more than one file? Or write more than one file? Or delete files, or open sockets, or sleep for a specified time, . . . ? The next alternative, and one actually adopted by the first version of Haskell, is to enrich the argument and result type of the main function:

```
main :: [Response] -> [Request]
```

Now the program takes as its argument a (lazy) list of `Response` values and produces a (lazy) list of `Request` values (Figure 1). Informally a `Request` says something like "please get the contents of file

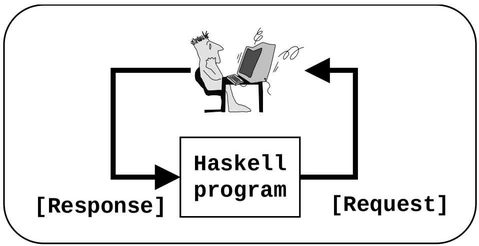


Figure 1: The stream I/O model

`/etc/motd`", while a `Response` might say "the contents you wanted is `No email today`". More concretely, `Request` and `Response` are both ordinary algebraic data types, something like this:

```
type FilePath = String

data Request = ReadFile FilePath
             | WriteFile FilePath String
             | ....

data Response = RequestFailed
              | ReadSucceeded String
              | WriteSucceeded
              | ...
```

There is still a wrapper program, as before. It repeatedly takes a request off the result list, acts on the request, and attaches an appropriate response to the argument list. There has to be some clever footwork to deal with the fact that the function has to be applied to a list of responses before there *are* any responses in the list, but that isn't a problem in a lazy setting.

This request/response story is expressive enough that it was adopted as the main input/output model in the first version of Haskell, but it has several defects:

- It is hard to extend. New input or output facilities can be added only by extending the `Request` and `Response` types, and by changing the "wrapper" program. Ordinary users are unlikely to be able to do this.
- There is no very close connection between a request and its corresponding response. It is extremely easy to write a program that gets one or more "out of step".

- Even if the program remains in step, it is easy to accidentally evaluate the response stream too eagerly, and thereby block emitting a request until the response to that request has arrived – which it won't.

Rather than elaborate on these shortcomings, we move swiftly on to a better solution, namely *monadic I/O*. Hudak and Sundaresh give a useful survey of approaches to purely-functional input/output [15], which describes the pre-monadic state of play.

2.2 Monadic I/O

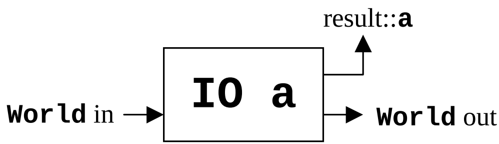
The big breakthrough in input/output for purely-functional languages came when we learned how to use so-called *monads* as a general structuring mechanism for functional programs. Here is the key idea:

> A value of type `IO a` is an "action" that, when performed, may do some input/output, before delivering a value of type `a`.

This is an admirably abstract statement, and I would not be surprised if it means almost nothing to you at the moment. So here is another, more concrete way of looking at these "actions":

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

This type definition says that a value of type `IO a` is a function that, when applied to an argument of type `World`, delivers a new `World` together with a result of type `a`. The idea is rather program-centric: the program takes the state of the entire world as its input, and delivers a modified world as a result, modified by the effects of running the program. I will say in Section 3.1 why I don't think this view of `IO` actions as functions is entirely satisfactory, but it generates many of the right intuitions, so I will use it unashamedly for a while. We may visualise a value of type `IO a` like this:

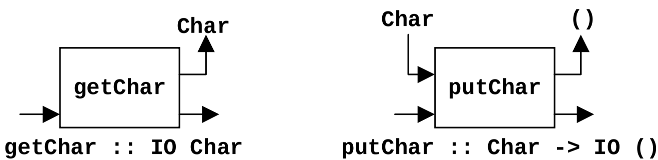


The `World` is fed in on the left, while the new `World`, and the result of type `a`, emerge on the right. In general, we will call a value of type `IO a` an *I/O action* or just *action*. In the literature you will often also find them called *computations*.

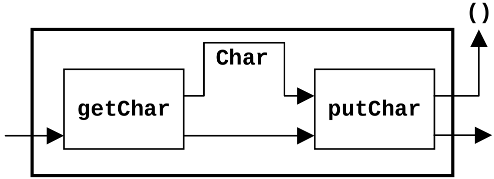
We can give `IO` types to some familiar operations, which are supplied as primitive:

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

`getChar` is an I/O action that, when performed, reads a character from the standard input (thereby having an effect on the world outside the program), and returns it to the program as the result of the action. `putChar` is a function that takes a character and returns an action that, when performed, prints the character on the standard output (its effect on the external world), and returns the trivial value `()`. The pictures for these actions look like this (the box for `putChar` takes an extra input for the `Char` argument):



Suppose we want to read a character, and print the character we have read. Then we need to glue together `putChar` and `getChar` into a compound action, like this:



`getChar >>= putChar`

To achieve this we use a glue function, or combinator, also provided as primitive:

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



```
echo :: IO ()
echo = getChar >>= putChar
```

The combinator `(>>=)` is often pronounced "bind". It implements sequential composition: it passes the result of performing the first action to the (parameterised) second action. More precisely, when the compound action `(a >>= f)` is performed, it performs action `a`, takes the result, applies `f` to it to get a new action, and then performs that new action. In the `echo` example, `(getChar >>= putChar)` first performs the action `getChar`, yielding a character c, and then performs `putChar` c.

Suppose that we wanted to perform `echo` twice in succession. We can't say `(echo >>= echo)`, because `(>>=)` expects a *function* as its second argument, not an action. Indeed, we want to throw away the result, `()`, of the first `echo`. It is convenient to define a second glue combinator, `(>>)`, in terms of the first:

```
(>>) :: IO a -> IO b -> IO b
(>>) a1 a2 = a1 >>= (\x -> a2)
```

The term `(\x -> a2)` is Haskell's notation for a lambda abstraction. This particular abstraction simply consumes the argument, `x`, throws it away, and returns `a2`. Now we can write

```
echoTwice :: IO ()
echoTwice = echo >> echo
```

"`(>>)`" is often pronounced "then", so we can read the right hand side as "`echo` then `echo`".

In practice, it is very common for the second argument of `(>>=)` to be an explicit lambda abstraction. For example, here is how we could read a character and print it twice:

```
echoDup :: IO ()
echoDup = getChar >>= (\c -> (putChar c >> putChar c))
```

All the parentheses in this example are optional, because a lambda abstraction extends as far to the right as possible, and you will often see this laid out like this:

```
echoDup :: IO ()
echoDup = getChar    >>= \c ->
          putChar c  >>
          putChar c
```

The fact that this looks a bit like a sequence of imperative actions is no coincidence — that is exactly what we wish to specify. Indeed, in Section 2.3 we will introduce special syntax to mirror an imperative program even more closely.

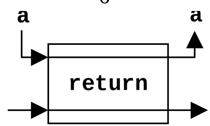
How could we write an I/O action that reads two characters, and returns both of them? We can start well enough:

```
getTwoChars :: IO (Char,Char)
getTwoChars = getChar    >>= \c1 ->
              getChar    >>= \c2 ->
              ???
```

But what are we to put for the "???" part? It must be of type `IO (Char,Char)`, but we have done all the input/output required. What we need is one more combinator:

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

The action `(return v)` is an action that does no I/O, and immediately returns `v` without having any side effects. We may draw its picture like this:



Now we can easily complete `getTwoChars`:

```
getTwoChars :: IO (Char,Char)
getTwoChars = getChar    >>= \c1 ->
              getChar    >>= \c2 ->
              return (c1,c2)
```

Here is a more realistic action that reads a whole line of input:

```
getLine :: IO [Char]
getLine = getChar     >>= \c ->
          if c == '\n' then
              return []
          else
              getLine     >>= \cs ->
              return (c : cs)
```

In this example, the "`[]`" is Haskell's notation for the empty list, while the infix constructor "`:`" is the list constructor.

A complete Haskell program defines a single big I/O action, called `main`, of type `IO ()`. The program is executed by performing the action. Here, for example, is a program that reads a complete line from the input, reverses it, and prints it on the output:

```
main :: IO ()
main = getLine        >>= \ cs ->
       putLine (reverse cs)
```

We have not yet defined `putLine :: [Char] -> IO ()`; we leave it as an exercise.

Notice that the only operation that combines, or composes I/O actions is `(>>=)`, *and it treats the world in a single-threaded way*. That is, it takes the world produced from the first action and passes it on to the second action. The world is never duplicated or thrown away, no matter what code the programmer writes. It is this property that allows us to implement `getChar` (and other `IO` primitives) by performing the operation right away — a sort of "update in place". I will say more about implementation in Section 2.8.

You might worry that there is an unbounded number of possible I/O "primitives", such as `putChar` and `getChar`, and you would be right. Some operations can be defined in terms of existing ones (such as `getLine`) but many cannot. What is needed, of course, is a way to call arbitrary I/O libraries supplied by the operating system, a topic I discuss in detail in Section 6.

2.3 “do” notation

Rather than make you write programs in the stylised form of the last section, Haskell provides a special syntax, dubbed “the `do` notation”, for monadic computations. Using the `do` notation we can write `getTwoChars` as follows:

```
getTwoChars :: IO (Char,Char)
getTwoChars = do { c1 <- getChar ;
                   c2 <- getChar ;
                   return (c1,c2)
                 }
```

You can leave out the "`c <-`" part when you want to throw away the result of the action:

```
putTwoChars :: (Char,Char) -> IO ()
putTwoChars (c1,c2) = do { putChar c1; putChar c2 }
```

The syntax is much more convenient than using `(>>=)` and lambdas, so in practice everyone uses `do` notation for I/O-intensive programs in Haskell. But it is just notation! The compiler translates the `do` notation into calls to `(>>=)`, just as before. The translation rules are simple[1]:

$$
\begin{array}{rcl}
\texttt{do } \{ x \texttt{ <- } e ; s \} & = & e \texttt{ >>= } \backslash x \texttt{ -> do } \{ s \} \\
\texttt{do } \{ e ; s \} & = & e \texttt{ >> do } \{ s \} \\
\texttt{do } \{ e \} & = & e
\end{array}
$$

It follows from this translation that the `do` statement "`x <- e`" *binds the variable* `x`. It does not *assign to the location* `x`, as would be the case in an imperative program. If we use the same variable name twice on the left hand side, we bind two distinct variables. For example:

```
do { c <- getChar ;    -- c :: Char
     c <- putChar c ; -- c :: ()
     return c
   }
```

The first line binds `c` to the character returned by `getChar`. The second line feeds that `c` to `putChar` and binds a *distinct* `c` to the value returned by `putChar`, namely `()`. This example also demonstrates that the scope of `x` bound by "`x <- e`" does not include `e`.

A `do` expression can appear anywhere that an expression can (as long as it is correctly typed). Here, for example, is `getLine` in `do` notation; it uses a nested `do` expression:

```
getLine :: IO [Char]
getLine = do { c <- getChar ;
               if c == '\n' then
```

```
                return []
              else
                do { cs <- getLine ;
                     return (c:cs)
         }      }
```

2.4 Control structures

If monadic I/O lets us do imperative programming, what corresponds to the control structures of imperative languages: for-loops, while-loops, and so on? In fact, we do not need to add anything further to get them: we can build them out of functions.

[1] Haskell also allows a `let` form in `do` notation, but we omit that for brevity.

For example, after some initialisation our web server goes into an infinite loop, awaiting service requests. We can easily express an infinite loop as a combinator:

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

So `(forever a)` is an action that repeats `a` forever; this iteration is achieved through the recursion of `forever`. Suppose instead that we want to repeat a given action a specified number of times. That is, we want a function:

```
repeatN :: Int -> IO a -> IO ()
```

So `(repeatN n a)` is an action that, when performed, will repeat `a` `n` times. It is easy to define:

```
repeatN 0 a = return ()
repeatN n a = a >> repeatN (n-1) a
```

Notice that `forever` and `repeatN`, like `(>>)` and `(>>=)`, take an action as one of their arguments. It is this ability to treat an action as a first class value that allows us to define our own control structures. Next, a `for` loop:

```
for :: [a] -> (a -> IO ()) -> IO ()
```

The idea is that `(for ns fa)` will apply the function `fa` to each element of `ns` in turn, in each case giving an action; these actions are then combined in sequence.

```
for []     fa = return ()
for (n:ns) fa = fa n >> for ns fa
```

We can use `for` to print the numbers between 1 and 10, thus:

```
printNums = for [1..10] print
```

(Here, `[1..10]` is Haskell notation for the list of integers between 1 and 10; and `print` has type `Int -> IO ()`.) Another way to define `for` is this:

```
for ns fa = sequence_ (map fa ns)
```

Here, `map` applies `fa` to each element of `ns`, giving a list of actions; then `sequence_` combines these actions together in sequence. So `sequence_` has the type

```
sequence_ :: [IO a] -> IO ()
sequence_ as = foldr (>>) (return ()) as
```

The "_" in "`sequence_`" reminds us that it throws away the results of the sub-actions, returning only `()`. We call this function "`sequence_`" because it has a close cousin, with an even more beautiful type:

```
sequence :: [IO a] -> IO [a]
```

It takes a list of actions, each returning a result of type `a`, and glues them together into a single compound action returning a result of type `[a]`. It is easily defined:

```
sequence []     = return []
sequence (a:as) = do { r  <- a;
                       rs <- sequence as ;
                       return (r:rs) }
```

Notice what is happening here. Instead of having a fixed collection of control structures provided by the language designer, we are free to invent new ones, perhaps application-specific, as the need arises. This is an extremely powerful technique.

2.5 References

The `IO` operations so far allow us to write programs that do input/output in strictly-sequentialised, imperative fashion. It is natural to ask whether we can also model another pervasive feature of imperative languages, namely mutable variables. Taking inspiration from ML's `ref` types, we can proceed like this:

```
data IORef a  -- An abstract type
newIORef   :: a -> IO (IORef a)
readIORef  :: IORef a -> IO a
writeIORef :: IORef a -> a -> IO ()
```

A value of type `IORef a` is a reference to a mutable cell holding a value of type `a`. A new cell can be allocated using `newIORef`, supplying an initial value. Cells can be read and written using `readIORef` and `writeIORef`.

Here is a small loop to compute the sum of the values between `1` and `n` in an imperative style:

```
count :: Int -> IO Int
count n = do { r <- newIORef 0 ;
               loop r 1 }
        where
          loop :: IORef Int -> Int -> IO Int
          loop r i | i>n       = readIORef r
                   | otherwise = do { v <- readIORef r ;
                                      writeIORef r (v+i) ;
                                      loop r (i+1) }
```

Just for comparison, here is what it might look like in C:

```
count( int n ) {
  int i, v = 0 ;
  for (i=1; i<=n; i++) { v = v+i ; }
  return( v ) ;
}
```

But this is an absolutely terrible example! For a start, the program is much longer and clumsier than it would be in a purely-functional style (e.g. simply `sum [1..n]`). Moreover, it purports to need the `IO` monad but does not really require any side effects at all. Thus, the `IO` monad enables us to transliterate an imperative program into Haskell, but if that's what you want to do, it would be better to use an imperative language in the first place!

Nevertheless, an `IORef` is often useful to "track" the state of some external-world object. For example, Haskell 98 provides a direct analogy of the Standard C library functions for opening, reading, and writing a file:

```
openFile :: String -> IOMode -> IO Handle
hPutStr  :: Handle -> [Char] -> IO ()
hGetLine :: Handle -> IO [Char]
hClose   :: Handle -> IO ()
```

Now, suppose you wanted to record how many characters were read or written to a file. A convenient way to do this is to arrange that `hPutStr` and `hGetLine` each increment a mutable variable suitably. The `IORef` can be held in a modified Handle:

```
type HandleC = (Handle, IORef Int)
```

Now we can define a variant of `openFile` that creates a mutable variable as well as opening the file, returning a `HandleC`; and variants of `hPutStr` and `hGetLine` that take a `HandleC` and modify the mutable variable appropriately. For example:


```
openFileC :: String -> IOMode -> IO HandleC
openFileC fn mode = do { h <- openFile fn mode ;
                         v <- newIORef 0 ;
                         return (h,v) }

hPutStrC :: HandleC -> String -> IO ()
hPutStrC (h,r) cs = do { v <- readIORef r ;
                         writeIORef r (v + length cs) ;
                         hPutStr h cs }
```

In this example, the mutable variable models (part of) the state of the file being written to, by tracking the number of characters written to the file. Since the file itself is, in effect, an external mutable variable, it is not surprising that an internal mutable variable is appropriate to model its state.

2.6 Leaving the safety belt at home

I have been careful to introduce the `IO` monad as an *abstract data type*: that is, a type together with a collection of operations over that type. In particular, we have:

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

getChar :: IO Char
putChar :: Char -> IO ()
...more operations on characters...

openFile :: [Char] -> IOMode -> IO Handle
...more operations on files...

newIORef :: a -> IO (IORef a)
...more operations on IORefs...
```

A key feature of an abstract data type is what it *prevents* as well as what it *permits*. In particular, notice the following:

- All the operations except one, `(>>=)`, have an I/O action as their *result*, but do not take one as an *argument*.

- The only operation that *combines* I/O actions is `(>>=)`.
- The `IO` monad is "sticky": no operation takes argument(s) with an `IO` type and returns a result with a non-`IO` type.

Sometimes, however, such restrictions are irksome. For example, suppose you wanted to read a configuration file to get some options for your program, using code something like this:

```
configFileContents :: [String]
configFileContents = lines (readFile "config")        -- WRONG!

useOptimisation :: Bool
useOptimisation = "optimise" `elem` configFileContents
```

Here, `lines :: String -> [String]` is a standard function that breaks a string into its constituent lines, while `elem :: Eq a => a -> [a] -> Bool` tells whether its first argument is a member of its second argument. Alas, the code is not type correct, because `readFile` has type

```
readFile :: FilePath -> IO String
```


So `readFile` produces an `IO String`, while `lines` consumes a `String`. We can "solve" this by giving `configFileContents` the type `IO String`, and `useOptimisation` the type `IO Bool`, plus some changes to the code. But that means we can only test `useOptimisation` when we are in the `IO` monad[2], which would be very inconvenient! What we want is a way to get from `IO String` to `String`, *but that is the very thing we cannot do in the* `IO` *monad*!

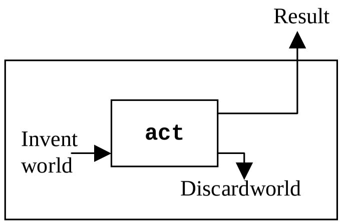
There is a good reason for this: reading a file is an I/O action, so in principle it matters *when* we read the file, relative to all the other I/O operations in the program. But in this case, we are confident that the file `config` will not change during the program run, so it really doesn't matter when we read it. This sort of thing happens often enough that all Haskell implementations offer one more, unsafe, I/O primitive:

```
unsafePerformIO :: IO a -> a
```

Now we can write

```
configFileContents :: [String]
configFileContents = lines (unsafePerformIO (readFile "config"))
```

and all is well. This combinator has a deliberately long name! Whenever you use it, you are promising the compiler that the timing of this I/O operation, relative to all the other I/O operations of the program, does not matter. You must undertake this proof obligation, because the compiler cannot do it for you; that is what the "`unsafe`" prefix means. Just to make the point even clearer, here is the "plumbing diagram" for `unsafePerformIO`:



`unsafePerformIO act`

As you can see, we have to invent a world out of thin air, and then discard it afterwards.

`unsafePerformIO` is a dangerous weapon, and I advise you against using it extensively. `unsafePerformIO` is best regarded as a tool for systems programmers and library writers, rather than for casual programmers. Because the input/output it encapsulates can happen at unpredictable moments (or even not at all) you need to know what you are doing. What is less obvious is that you can also use it to defeat the Haskell type, by writing a function `cast :: a -> b`; see [25]!

`unsafePerformIO` is often mis-used to force an imperative program into a purely-functional setting. This a bit like using a using a chain saw to repair a dishwasher — it's the wrong tool for the job. Such programs can invariably be restructured into a cleaner, functional form. Nevertheless, when the proof obligations are satisfied, `unsafePerformIO` can be extremely useful. In practice, I have encountered three very common patterns of usage:

- Performing once-per-run input/output, as for `configFileContents`.
- Allocating a global mutable variable. For example:

```
noOfOpenFiles :: IORef Int
noOfOpenFiles = unsafePerformIO (newIORef 0)
```

- Emitting trace messages for debugging purposes:

[2] We would also need to be careful not to read the file every time we tested the boolean!


```
trace :: String -> a -> a
trace s x = unsafePerformIO (putStrLn s >> return x)
```

2.7 A quick review

Let us summarise what we have learned so far:

- A complete Haskell program is a single (perhaps large) I/O action called `main`.
- Big I/O actions are built by gluing together smaller actions using `(>>=)` and `return`.
- An I/O action is a first-class value. It can be passed to a function as an argument, or returned as the result of a function call (consider `(>>)`, for example). Similarly, it can be stored in a data structure — consider the argument to `sequence`, for example.
- The fact that I/O actions can be passed around so freely makes it easy to define new "glue" combinators in terms of existing ones.

Monads were originally invented in a branch of mathematics called category theory, which is increasingly being applied to describe the semantics of programming languages. Eugenio Moggi first identified the usefulness of monads to describe composable "computations" [32]. Moggi's work, while brilliant, is not for the faint hearted. For practical programmers the breakthrough came in Phil Wadler's paper "Comprehending monads" [47], in which he described the usefulness of monads in a programming context. Wadler wrote several more very readable papers about monads, which I highly recommend [48, 49, 50]. He and I built directly on this work to write the first paper about monadic I/O [38].

In general, a *monad* is a triple of a type constructor M, and two functions, `return` and `>>=`, with types

$$\begin{array}{rcl} \texttt{return} & :: & \forall\alpha.\ \alpha \rightarrow M\ \alpha \\ \texttt{>>=} & :: & \forall\alpha\beta.\ M\ \alpha \rightarrow (\alpha \rightarrow M\ \beta) \rightarrow M\ \beta \end{array}$$

That is not quite all: these three must satisfy the following algebraic laws:

$$\texttt{return}\, x \mathbin{\texttt{>>=}} f \quad = \quad f\ x \qquad (LUNIT)$$

$$m \mathbin{\texttt{>>=}} \texttt{return} \quad = \quad m \qquad (RUNIT)$$

$$\frac{x \text{ does not appear free in } m_3}{m_1 \mathbin{\texttt{>>=}} (\lambda x.m_2 \mathbin{\texttt{>>=}} (\lambda y.m_3)) \quad = \quad (m_1 \mathbin{\texttt{>>=}} (\lambda x.m_2)) \mathbin{\texttt{>>=}} (\lambda y.m_3)} \; (BIND)$$

(In this box and ones like it, I use names like (LUNIT) simply as a convenient way to refer to laws from the running text.) The last of these rules, (BIND), is much easier to understand when written in `do` notation:

```
do { x <- m1;            do { y <- do { x <- m1;
     y <- m2;      =                     m2}
     m3                       m3}
```

In any correct implementation of the `IO` monad, `return` and `(>>=)` should satisfy these properties. In these notes I present only one monad, the `IO` monad, but a single program may make use of many different monads, each with its own type constructor, $return$ and $bind$ operators. Haskell's type class mechanism allows one to overload the functions `return` and `(>>=)` so they can be used in any monad, and the `do` notation can likewise be used for any monad. Wadler's papers, cited above, give many examples of other monads, but we do not have space to pursue that topic here.

2.8 Implementation notes

How difficult is it for a compiler-writer to implement the `IO` monad? There seem to be two main alternatives.

Keep the monad right through. The first technique carries the `IO` monad right through the compiler to the code generator. Most functional-language compilers translate the source program to an intermediate form based closely on the lambda calculus, apply optimising transformations to that intermediate form, and then generate code. It is entirely possible to extend the intermediate form by adding monadic constructs. One could simply add `(>>=)` and `return` as primitives, but it makes transformation much easier if one adds the `do`-notation directly, instead of a primitive `(>>=)` function. (Compare the two forms of the (BIND) rule given in the previous section.) This is the approach taken by Benton and Kennedy in MLj, their implementation of ML [6].

The functional encoding. The second approach, and the one used in the Glasgow Haskell Compiler (GHC), is to adopt the functional viewpoint of the `IO` monad, which formed the basis of our earlier pictorial descriptions:

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

If we represent the "world" argument by an un-forgeable token, of type `World`, then we can directly implement `return` and `(>>=)` like this:

```
return :: a -> IO a
return a = \w -> (a,w)

(>>=) :: IO a -> (a -> IO b) -> IO b
(>>=) m k = \w -> case m w of
                    (r,w') -> k r w'
```

Here `w` is the un-forgeable token that stands for the world. In the definition of `(>>=)` we see that the world returned by the first action is passed to the second, just as in the picture in Section 2.2. We must also implement the primitive `IO` operations, such as `getChar`, but that is now no different to implementing other primitive operations, such as addition of two integers.

So which of these two approaches is better? Keeping the `IO` monad explicit is principled, but it means that every optimisation pass must deal explicitly with the new constructs. GHC's approach is more economical. For example, the three laws in Section 2.7, regarded as optimisations, are simple consequences and need no special attention. All the same, I have to say that I think the GHC approach is a bit of a hack. Why? Because it relies for its correctness on the fact that the compiler never duplicates a redex. Consider this expression:

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

If we use GHC's definitions of `(>>=)` we can translate this to:

```
\w -> case getChar w of
        (c,w1) -> case putChar c w1 of
                    (_,w2) -> putChar c w2
```

The compiler would be entirely justified in replacing this code with:

```
\w -> case getChar w of
        (c,w1) -> case putChar c w1 of
                    (_,w2) -> putChar (fst (getChar w)) w2
```


Here I have replaced the second use of `c` with another call to `getChar w`. Two bad things have happened: first, the incoming world token, `w`, has been duplicated; and second, there will now be two calls to `getChar` instead of one. If this happens, our assumption of single-threadedness no longer holds, and neither does our efficient "update-in-place" implementation of `getChar`. Catastrophe!

In the functional language Clean, the whole I/O system is built on an explicit world-passing style. The single-threadedness of the world is ensured by Clean's uniqueness-type system, which verifies that values which should be single-threaded (notably the world) are indeed used in single threaded way [4]. In Haskell, the `IO` monad maintains the world's single-threadedness by construction; so the programmer cannot err, but it is in principle possible for the compiler to do so.

In practice, GHC is careful never to duplicate an expression whose duplication might give rise to extra work (a redex), so it will never duplicate the call to `getChar` in this way. Indeed, Ariola and Sabry have shown formally that if the compiler never duplicates redexes, then indeed our implementation strategy is safe [2]. So GHC's approach is sound, but it is uncomfortable that an apparently semantics-preserving transformation, such as that above, does not preserve the semantics at all. This observation leads us neatly to the next question I want to discuss, namely how to give a semantics to the Awkward Squad.

3 What does it all mean?

It is always a good thing to give a precise semantics to a language feature. How, then, can we give a semantics for the `IO` monad? In this section I will describe the best way I know to answer this question. I will introduce notation as we go, so you should not need any prior experience of operational semantics to understand this section. You can also safely skip to Section 4. Nevertheless, I urge to persevere, because I will use the same formal framework later, to explain the semantics of concurrency and exceptions.

3.1 A denotational semantics?

One approach to semantics is to take the functional viewpoint I described earlier:

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

In this view, the meaning of an action is just a function. One can make this story work, but it is a bit unsatisfactory:

- Regarded as a function on `World`s, this program

  ```
  loop :: IO ()
  loop = loop
  ```

  has denotation bottom ($\perp$). But, alas, this program

  ```
  loopX :: IO ()
  loopX = putChar 'x' >> loopX
  ```

  unfortunately also has denotation $\perp$. Yet these programs would be regarded as highly distinguishable by a user (one loops for ever, the other prints `'x'` for ever). Nor is the problem restricted to erroneous programs: some programs (server processes, for example) may be *designed* to run essentially forever, and it seems wrong to say that their meaning is simply $\perp$!

- Consider two Haskell programs running in parallel, each sending output to the other — a Web server and a Web browser, for example. The output of each must form part of the `World` given as the input to the other. Maybe it would be possible to deal with this through a fixpoint operator, but it seems complicated and un-intuitive (to me anyway!).

$$
\begin{array}{lrcl}
 & x, y & \in & \mathit{Variable} \\
 & k & \in & \mathit{Constant} \\
 & \mathit{con} & \in & \mathit{Constructor} \\
 & c & \in & \mathit{Char} \\
 & & & \\
\text{Values} & V & ::= & \backslash x\ \texttt{->}\ M \mid k \mid \mathit{con}\, M_1 \cdots M_n \mid c \\
 & & \mid & \texttt{return}\, M \mid M \texttt{ >>= } N \\
 & & \mid & \texttt{putChar}\, c \mid \texttt{getChar} \\
 & & & \\
\text{Terms} & M, N, H & ::= & x \mid V \mid M\,N \mid \texttt{if}\ M\ \texttt{then}\ N_1\ \texttt{else}\ N_2 \mid \cdots \\
 & & & \\
\text{Evaluation contexts} & \mathbb{E} & ::= & [\cdot] \mid \mathbb{E} \texttt{ >>= } M
\end{array}
$$

Figure 2: The syntax of values and terms.

- The approach does not scale well when we add concurrency, which we will do in Section 4.

These problems may be soluble while remaining in a denotational framework, perhaps by producing a sequence of `World`s, or by returning a set of *traces* rather than a new `World`. To give the idea of the trace approach, we model `IO` like this:

```
type IO a = (a, Set Trace)
type Trace = [Event]
data Event = PutChar Char | GetChar Char | ...
```

A program that reads one character, and echoes it back to the screen, would have semantics

```
((), { [GetChar 'a', PutChar 'a'],
       [GetChar 'b', PutChar 'b'],
       [GetChar 'c', PutChar 'c'],
       ... })
```

We return a *set* of traces, because the trace contains details of *inputs* as well as *outputs*, so there must be a trace for each possible input. The set of traces describes all the behaviours the program can have, and no others. For example `[GetChar 'x', PutChar 'y']` is excluded.

This approach is used to give the semantics of CSP by Roscoe [42]. However we will instead adopt an *operational* semantics, based on standard approaches to the semantics of process calculi [31]. Ultimately, I think the two approaches have similar power, but I find the operational approach simpler and easier to understand.

3.2 An operational semantics

Our semantics is stratified in two levels: an *inner denotational semantics* that describes the behaviour of pure terms, while an *outer monadic transition semantics* describes the behaviour of `IO` computations. We consider a simplified version of Haskell: our language has the usual features of a lazy functional language (lambda abstraction, application, data structures, case expressions, *etc.*), augmented with constants corresponding to `IO` operations. We will only present those elements of the syntax that are relevant to the semantics; other aspects (such as how we represent lists, or how to write a case expression) would not aid comprehension of the semantics, and are not presented.

M and N range over *terms* in our language, and V ranges over values (Figure 2). A *value* is a term that is considered by the inner, purely-functional semantics to be evaluated. The values in Figure 2 include constants and lambda abstractions, as usual, but they are unusual in two ways:

- *We treat the primitive monadic* `IO` *operations as values*. For example, `putChar 'c'` is a value. No further work can be done on this term in the purely-functional world; it is time to hand it over to the outer, monadic semantics. In the same way, M `>>=` N, `getChar`, and `return` M are all values.

- Some of these monadic `IO` values have arguments that are not arbitrary terms (M, N, *etc.*), but are themselves values (e.g. c). The only example in Figure 2 is the value `putChar` c but others will appear later. So `putChar 'A'` is a value, but `putChar (chr 65)` is not (it is a term, though). It is as if `putChar` is a *strict* data constructor. The reason for this choice is that evaluating `putChar`'s argument is something that can be done in the purely-functional world; indeed, it *must* be done before the output operation can take place.

We will give the semantics by describing how one *program state* evolves into a new program state by making a *transition*. For now, we model a program state simply as a term, but we write it in curly braces, thus $\{M\}$, to remind us that it is a program state.

3.3 Labelled transitions

The transition from one program state to the next may or may not be *labelled* by an *event*, α. So we write a transition like this:

$$P \xrightarrow{\alpha} Q$$

The events α represent communication with the external environment; that is, input and output. Initially we will use just two events:

- $P \xrightarrow{!c} Q$ means "program state P can move to Q, by writing the character c to the standard output".

- $P \xrightarrow{?c} Q$ means "program state P can move to Q, by reading the character c from the standard input".

Here, then, are our first two transition rules.

$$\begin{array}{rcl} \{\texttt{putChar}\ c\} & \xrightarrow{!c} & \{\texttt{return ()}\} \\ \{\texttt{getChar}\} & \xrightarrow{?c} & \{\texttt{return}\ c\} \end{array}$$

The first rule says that a program consisting only of `putChar` c can make a transition, labelled by $!\,c$, to a program consisting of `return ()`. The second rule is similar. But most programs consist of more than a single I/O action! What are we to do then? To answer that question we introduce *evaluation contexts*.

3.4 Evaluation contexts

The `getChar` transition rule is all very well, but what if the program consists of more than a single `getChar`? For example, consider the program[3]:

```
main = getChar >>= \c -> putChar (toUpper c)
```

Which is the first I/O action that should be performed? The `getChar`, of course! We need a way to say "the first I/O action to perform is to the left of the `(>>=)`". Sometimes we may have to look to the left of more than one `(>>=)`. Consider the slightly artificial program

```
main = (getChar >>= \c -> getChar) >>= \d -> return ()
```

[3] `toUpper :: Char -> Char` converts a lower-case character to upper case, and leaves other characters unchanged.

$$\{\mathbb{E}[\texttt{putChar}\ c]\} \xrightarrow{!c} \{\mathbb{E}[\texttt{return ()}]\} \quad (PUTC)$$

$$\{\mathbb{E}[\texttt{getChar}]\} \xrightarrow{?c} \{\mathbb{E}[\texttt{return}\ c]\} \quad (GETC)$$

$$\{\mathbb{E}[\texttt{return}\ N \texttt{ >>= } M]\} \rightarrow \{\mathbb{E}[M\ N]\} \quad (LUNIT)$$

$$\frac{\mathcal{E}[\![M]\!] = V \quad M \not\equiv V}{\{\mathbb{E}[M]\} \rightarrow \{\mathbb{E}[V]\}} \ (FUN)$$

Figure 3: The basic transition rules

Here, the first I/O action to be performed is the leftmost `getChar`. In general, to find the first I/O action we "look down the left branch of the tree of `(>>=)` nodes".

We can formalise all this arm-waving by using the now well-established notion of an *evaluation context* [9, 52]. The syntax of evaluation contexts is this (Figure 2):

$$\mathbb{E} \quad ::= \quad [\cdot] \mid \mathbb{E} \texttt{ >>= } M$$

An evaluation context $\mathbb{E}$ is a term with a hole, written $[\cdot]$, in it. For example, here are three possible evaluation contexts:

$$\begin{array}{lcl} \mathbb{E}_1 & = & [\cdot] \\ \mathbb{E}_2 & = & [\cdot] \texttt{ >>= (\textbackslash c -> return (ord c))} \\ \mathbb{E}_3 & = & \texttt{(}[\cdot]\texttt{ >>= f) >>= g} \end{array}$$

In each case the "$[\cdot]$" indicates the location of the hole in the expression. We write $\mathbb{E}[M]$ to denote the result of filling the hole in $\mathbb{E}$ with the term M. Here are various ways of filling the holes in our examples:

$$\begin{array}{lcl} \mathbb{E}_1[\texttt{print "hello"}] & = & \texttt{print "hello"} \\ \mathbb{E}_2[\texttt{getChar}] & = & \texttt{getChar >>= (\textbackslash c -> return (ord c))} \\ \mathbb{E}_3[\texttt{newIORef True}] & = & \texttt{(newIORef True >>= f) >>= g} \end{array}$$

Using the notation of evaluation contexts, we can give the real rules for `putChar` and `getChar`, in Figure 3. In general we will give each transition rule in a figure, and give it a name — such as (PUTC) and (GETC) — for easy reference.

The rule for (PUTC), for example, should be read: "if a `putChar` occurs as the next I/O action, in a context $\mathbb{E}[\cdot]$, the program can make a transition, emitting a character and replacing the call to `putChar` by `return ()`". This holds for any evaluation context $\mathbb{E}[\cdot]$.

Let us see how to make transitions using our example program:

```
main = getChar >>= \c -> putChar (toUpper c)
```

Using rule (GETC) and the evaluation context ($[\cdot]$ `>>= \c -> putChar (toUpper c)`), and assuming that the environment delivers the character `'w'` in response to the `getChar`, we can make the transition:

$$\begin{array}{cl} & \{\texttt{getChar >>= \textbackslash c -> putChar (toUpper c)}\} \\ \xrightarrow{\texttt{?'w'}} & \\ & \{\texttt{return 'w' >>= \textbackslash c -> putChar (toUpper c)}\} \end{array}$$

How did we choose the correct evaluation context? The best way to see is to try choosing another one! The context we chose is the only one formed by the syntax in Figure 2 that allows any transition rule to fire. For example the context $[\cdot]$, which is certainly well-formed, would force the term in the hole to be `getChar >>= \c -> putChar (toUpper c)`, and no rule matches that. The context simply reaches down the left-branching chain of `(>>=)` combinators to reach the left-most action that is ready to execute.

What next? We use the (LUNIT) law of Section 2.7, expressed as a new transition rule:

$$\{\mathbb{E}[\texttt{return}\, N \texttt{>>=} M]\} \quad \rightarrow \quad \{\mathbb{E}[M\, N]\} \quad (LUNIT)$$

Using this rule, we make the transition

$$\begin{array}{ll} & \{\texttt{return 'w' >>= \textbackslash c -> putChar (toUpper c)}\} \\ \rightarrow & \\ & \{\texttt{(\textbackslash c -> putChar (toUpper c)) 'w'}\} \end{array}$$

Now we need to do some ordinary, purely-functional evaluation work. We express this by "lifting" the inner denotational semantics into our transition system, like this (the "(FUN)" stands for "functional"):

$$\frac{\mathcal{E}[\![M]\!] = V \qquad\qquad M \not\equiv V}{\{\mathbb{E}[M]\} \;\rightarrow\; \{\mathbb{E}[V]\}} \quad (FUN)$$

That is, if the term M has value V, as computed by the denotational semantics of M, namely $\mathcal{E}[\![M]\!]$, then we can replace M by V at the active site. The function $\mathcal{E}[\![]\!]$ is a mathematical function that given a term M, returns its value $\mathcal{E}[\![M]\!]$. This function defines the semantics of the *purely-functional* part of the language – indeed, $\mathcal{E}[\![]\!]$ is called the *denotational semantics* of the language. Denotational semantics is well described in many books [43, 1], so we will not study it here; meanwhile, you can simply think of $\mathcal{E}[\![M]\!]$ as the value obtained by evaluating M[4].

The side condition $M \not\equiv V$ is just there to prevent the rule firing repeatedly without making progress, because $\mathcal{E}[\![V]\!] = V$ for any V. Rule (FUN) allows us to make the following transition, using normal beta reduction:

$$\{(\texttt{\textbackslash c -> putChar (toUpper c)) 'w'}\} \rightarrow \{\texttt{putChar 'W'}\}$$

In making this transition, notice that $\mathcal{E}[\![]\!]$ produced the value `putChar 'W'`, and *not* `putChar (toUpper 'w')`. As we discussed towards the end of Section 3.2, we model `putChar` as a *strict* constructor.

Now we can use the `putChar` rule to emit the character:

$$\{\texttt{putChar 'W'}\} \xrightarrow{\texttt{!'W'}} \{\texttt{return ()}\}$$

And now the program is finished.

Referring back to the difficulties identified in Section 3.1, we can now distinguish a program `loop` that simply loops forever, from program `loopX` that repeatedly prints `'x'` forever. These programs both have denotation $\perp$ in a (simple) denotational semantics (Section 3.1), but they have different behaviours in our operational semantics. `loopX` will repeatedly make a transition with the label `!x`. But what happens to `loop`? To put it another way, what happens in rule (FUN) if $\mathcal{E}[\![M]\!] = \perp$? The simplest thing to say is that then there is no value V such that $\mathcal{E}[\![M]\!] = V$, and so (FUN) cannot fire. So no rule applies, and the program is stuck. This constitutes an observably different sequence of transitions than `loopX`[5].

Lastly, before we leave the topic of evaluation contexts, let us note that the term M in rule (FUN) always has type `IO` τ for some type τ; that is, an evaluation context $\mathbb{E}[\cdot]$ always has an I/O action in its hole. (Why? Because the hole in an evaluation context is either the whole program, of type `IO ()`, or the left argument of a `(>>=)`, of type `IO` τ for some τ.) So there is no need to explain how the program (say) $\{\texttt{True}\}$ behaves, because it is ill-typed.

---

[4] I am being a bit sloppy here, because a denotational semantics yields a mathematical value, not a term in the original language, but in fact nothing important is being swept under the carpet here. From a technical point of view it may well be simpler, in the end, to adopt an operational semantics for the inner purely-functional part too, but that would be a distraction here. Notice, too, that the valuation function of a denotational semantics would usually have an environment, ρ. But the rule (FUN) only requires the value of a closed term, so the environment is empty.

[5] By "observable" I mean "observable *looking only at the labelled transitions*"; the labelled transitions constitute the interaction of the program with its environment. You may argue that we should not say that `loop` gets "stuck" when actually it is in an infinite

loop. For example, the program `forever (return ())` is also an infinite loop with no external interactions, and it makes an infinite sequence of (unlabelled) transitions. If you prefer, one can instead add a variant of (FUN) that makes an un-labelled transition to an unchanged state if $\mathcal{E}[\![M]\!] = \perp$. Then `loop` would also make an infinite sequence of un-labelled transitions. It's just a matter of taste.

$$
\begin{array}{rcll}
r & \in & \texttt{IORef} & \\
\\
V & ::= & \ldots \mid \texttt{writeIORef}\, r\, N \mid \texttt{readIORef}\, r \mid \texttt{newIORef}\, M \mid r & \\
\\
P, Q, R & ::= & \{M\} & \text{The main program} \\
 & \mid & \langle M \rangle_r & \text{An } \texttt{IORef} \text{ named } r\text{, holding } M \\
 & \mid & P \mid Q & \text{Parallel composition} \\
 & \mid & \nu x.P & \text{Restriction}
\end{array}
$$

$$
\begin{array}{rcll}
\{\mathbb{E}[\texttt{readIORef}\, r]\} \mid \langle M \rangle_r & \rightarrow & \{\mathbb{E}[\texttt{return}\, M]\} \mid \langle M \rangle_r & (\mathit{READIO}) \\
\{\mathbb{E}[\texttt{writeIORef}\, r\, N]\} \mid \langle M \rangle_r & \rightarrow & \{\mathbb{E}[\texttt{return ()}]\} \mid \langle N \rangle_r & (\mathit{WRITEIO})
\end{array}
$$

$$
\frac{r \notin \mathit{fn}\,(\mathbb{E}, M)}{\{\mathbb{E}[\texttt{newIORef}\, M]\} \rightarrow \nu r.(\{\mathbb{E}[\texttt{return}\, r]\} \mid \langle M \rangle_r)}\ (\mathit{NEWIO})
$$

Figure 4: Extensions for `IORef`s

3.5 Dealing with `IORefs`

Let us now add `IORefs` to our operational semantics. The modifications we need are given in Figure 4:

- We add a new sort of *value* for each `IORef` primitive; namely `newIORef`, `readIORef`, and `writeIORef`.
- We add a new sort of value for `IORef` identifiers, r. An `IORef` identifier is the value returned by `newIORef` — you can think of it as the address of the mutable cell.
- We extend a program state to be a main thread $\{M\}$, as before, together with zero or more `IORef`s, each associated with a reference identifier r.

The syntax for program states in Figure 4 might initially be surprising. We use a vertical bar to join the main thread and the `IORef`s into a program state. For example, here is a program state for a program that has (so far) allocated two `IORef`s, called r_1 and r_2 respectively:

$$\{M\} \mid \langle N_1 \rangle_{r_1} \mid \langle N_2 \rangle_{r_2}$$

If you like, you can think of running the (active) program M in parallel with two (passive) containers r_1 and r_2, containing N_1 and N_2 respectively.

Here is the rule for `newIORef`, which creates a new reference cell:

$$\frac{r \notin \mathit{fn}\,(\mathbb{E}, M)}{\{\mathbb{E}[\texttt{newIORef}\; M]\} \;\rightarrow\; \nu r.(\{\mathbb{E}[\texttt{return}\; r]\} \mid \langle M \rangle_r)} \; (\mathit{NEWIO})$$

If the next I/O action in the main program is to create a new `IORef`, then it makes a transition to a new state in which the main program is in parallel with a newly-created (and suitably initialised) `IORef` named r. What is r? It is an arbitrary name whose only constraint is that it must not already be used in M, or in the evaluation context $\mathbb{E}$. That is what the side condition $r \notin \mathit{fn}\,(\mathbb{E}, M)$ means — $\mathit{fn}\,(\mathbb{E}, M)$ means "the free names of $\mathbb{E}$ and M".

The new form $\nu r.P$, shown in the syntax of programs in Figure 4, means "let r be the name of a reference cell, in program state P". The ν is the binding for all the occurrences of r in P — it's always a good thing to know where a variable is bound. Another way to understand the side condition $r \notin fn\,(\mathbb{E}, M)$ is this: if the side condition did not hold, then the old occurrences of r in (say) M would erroneously be captured by the ν.

$$
\begin{array}{rcllr}
P \mid Q & \equiv & Q \mid P & & (COMM) \\
P \mid (Q \mid R) & \equiv & (P \mid Q) \mid R & & (ASSOC) \\
\nu x.\nu y.P & \equiv & \nu y.\nu x.P & & (SWAP) \\
(\nu x.P) \mid Q & \equiv & \nu x.(P \mid Q), & x \notin fn\,(Q) & (EXTRUDE) \\
\nu x.P & \equiv & \nu y.P[y/x], & y \notin fn\,(P) & (ALPHA)
\end{array}
$$

$$
\frac{P \xrightarrow{\alpha} Q}{P \mid R \xrightarrow{\alpha} Q \mid R}\ (PAR) \qquad \frac{P \xrightarrow{\alpha} Q}{\nu x.P \xrightarrow{\alpha} \nu x.Q}\ (NU)
$$

$$
\frac{P \equiv P' \quad P' \xrightarrow{\alpha} Q' \quad Q' \equiv Q}{P \xrightarrow{\alpha} Q}\ (EQUIV)
$$

Figure 5: Structural congruence, and structural transitions.

Here are the rules for reading and writing `IORefs`:

$$
\{\mathbb{E}[\texttt{readIORef}\ r]\} \mid \langle M\rangle_r \;\rightarrow\; \{\mathbb{E}[\texttt{return}\ M]\} \mid \langle M\rangle_r \qquad (READIO)
$$

$$\{\mathbb{E}[\texttt{writeIORef}\, r\, N]\} \mid \langle M\rangle_r \;\rightarrow\; \{\mathbb{E}[\texttt{return ()}]\} \mid \langle N\rangle_r \quad (\mathit{WRITEIO})$$

The rule for `readIORef` says that if the next I/O action in the main program is `readIORef` r, and the main program is parallel with an `IORef` named r containing M, then the action `readIORef` r can be replaced by `return` M[6]. This transition is quite similar to that for `getChar`, except that the transition is unlabelled because it is internal to the program — remember that only labelled transitions represent interaction with the external environment.

We have several tiresome details to fix up. First, we originally said that the transitions were for whole program states, but these two are for only *part* of a program state; there might be other `IORef`s, for example. Second, what if the main program was not adjacent to the relevant `IORef`? We want to say somehow that it can become adjacent to whichever `IORef` it pleases. To formalise these matters we have to give several "structural" rules, given in Figure 5. Rule (PAR), for example, says that if P can move to Q, then P in parallel with anything (R) can move to Q in parallel with the same anything — in short, non-participating pieces of the program state are unaffected. The equivalence rules (COMM), (ASSOC) say that $\mid$ is associative and commutative, while (EQUIV) says that we are free to use these equivalence rules to bring parts of the program state together. In these rules, we take α to range over both events, such as `!`c and `?`c, and also over the empty label. (In the literature, you will often see the empty event written τ.) Finally, (SWAP), (EXTRUDE), and (ALPHA) allow you to move ν's around so that they don't get in the way.

It's all a formal game. If you read papers about operational semantics you will see these rules over and over again, so it's worth becoming comfortable with them. They aren't optional though; if you want to conduct water-tight proofs about what can happen, it's important to specify the whole system in a formal way.

Here is an example of working through the semantics for the following program:

```
main = newIORef 0    >>= \ v ->
       readIORef v   >>= \ n ->
       writeIORef v (n+1)
```

The program allocates a new `IORef`, reads it, increments its contents and writes back the new value. The semantics works like this, where I have saved space by abbreviating "`newIORef`" to "`new`" and similarly for `readIORef` and `writeIORef`:

[6] The alert reader will notice that (READIO) duplicates the term M, and hence models *call-by-name* rather that *call-by-need*. It is straightforward to model call-by-need, by adding a *heap* to the operational semantics, as Launchbury first showed [24]. However, doing so adds extra notational clutter that is nothing do to with the main point of this tutorial. In this tutorial I take the simpler path of modelling call-by-name.

$$
\begin{array}{lll}
 & \{\texttt{new 0 >>= \textbackslash v -> read v >>= \textbackslash n -> write v (n+1)}\} & \\
\rightarrow & \nu r.(\{\texttt{return}\, r\, \texttt{>>= \textbackslash v -> read v >>= \textbackslash n -> write v (n+1)}\} \mid \langle \texttt{0} \rangle_r) & (NEWIO) \\
\rightarrow & \nu r.(\{\texttt{(\textbackslash v -> read v >>= \textbackslash n -> write v (n+1))}\, r\} \mid \langle \texttt{0} \rangle_r) & (LUNIT) \\
\rightarrow & \nu r.(\{\texttt{read}\, r\, \texttt{>>= \textbackslash n -> write}\, r\, \texttt{(n+1))}\} \mid \langle \texttt{0} \rangle_r) & (FUN) \\
\rightarrow & \nu r.(\{\texttt{return 0 >>= \textbackslash n -> write}\, r\, \texttt{(n+1))}\} \mid \langle \texttt{0} \rangle_r) & (READIO) \\
\rightarrow & \nu r.(\{\texttt{(\textbackslash n -> write}\, r\, \texttt{(n+1)) 0}\} \mid \langle \texttt{0} \rangle_r) & (LUNIT) \\
\rightarrow & \nu r.(\{\texttt{write}\, r\, \texttt{(0+1))}\} \mid \langle \texttt{0} \rangle_r) & (FUN) \\
\rightarrow & \nu r.(\{\texttt{return ()}\} \mid \langle \texttt{0+1} \rangle_r) & (WRITEIO)
\end{array}
$$

It should be clear that naming a new `IORef` with a name that is already in use would be a Bad Thing. That is the reason for the side condition on rule (NEWIO) says that r cannot be mentioned in $\mathbb{E}$ or M. But what if r was in use somewhere *else* in the program state — remember that there may be other threads running in parallel with the one we are considering? That is the purpose of the "νr" part: it restricts the scope of r. Having introduced ν in this way, we need a number of structural rules (Figure 5) to let us move ν around. Notably, (EXTRUDE) lets us move all the ν's to the outside. Before we can use (EXTRUDE), though, we may need to use (ALPHA) to change our mind about the name we chose if we come across a name-clash. Once all the ν's are at the outside, they don't get in the way at all.

4 Concurrency

A web server works by listening for connection requests on a particular socket. When it receives a request, it establishes a connection and engages in a bi-directional conversation with the client. Early versions of the HTTP protocol limited this conversation to one utterance in each direction ("please send me this page"; "ok, here it is"), but more recent versions of HTTP allow multiple exchanges to take place, and that is what we do here.

If a web server is to service multiple clients, it must deal concurrently with each client. It is simply not acceptable to deal with clients one at a time. The obvious thing to do is to fork a new *thread* of some kind for each new client. The server therefore must be a *concurrent* Haskell program.

I make a sharp distinction between *parallelism* and *concurrency*:

- A *parallel* functional program uses multiple processors to gain performance. For example, it may be faster to evaluate $e_1 + e_2$ by evaluating e_1 and e_2 in parallel, and then add the results. Parallelism has no semantic impact at all: the meaning of a program is unchanged whether it is executed sequentially or in parallel. Furthermore, the results are deterministic; there is no possibility that a parallel program will give one result in one run and a different result in a different run.

- In contrast, a *concurrent* program has concurrency as part of its specification. The program must run concurrent threads, each of which can independently perform input/output. The program may be run on many processors, or on one — that is an implementation choice. The behaviour of the program is, necessarily and by design, non-deterministic. Hence, unlike parallelism, concurrency has a substantial semantic impact.

Of these two, my focus in these notes is exclusively on concurrency, not parallelism. For those who are interested, a good introduction to parallel functional programming is [46], while a recent book gives a comprehensive coverage [12].

Concurrent Haskell [35] is an extension to Haskell 98 designed to support concurrent programming, and we turn next to its design.

4.1 Threads and `forkIO`

Here is the main loop of the web server:

```
acceptConnections :: Config -> Socket -> IO ()
acceptConnections config socket
  = forever (do { conn <- accept socket ;
                  forkIO (serviceConn config conn) })
```

(We defined `forever` in Section 2.4.) This infinite loop repeatedly calls `accept`, a Haskell function that calls the Unix procedure of the same name (*via* mechanisms we will discuss in Section 6), to accept a new connection. `accept` returns, as part of its result, a `Handle` that can be used to communicate with the client.

```
accept :: Socket -> IO Connection

type Connection = (Handle,    -- Read from here
```

```
                    SockAddr)  -- Peer details
```

Having established a connection, `acceptConnections` then uses `forkIO` to fork off a fresh thread, `(serviceConn config conn)`, to service that connection. The type of `forkIO` is this:

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

It takes an I/O action and arranges to run it concurrently with the "parent" thread. The call to `forkIO` returns immediately, returning as its result an identifier for the forked thread. We will see in Section 5.3 what this `ThreadId` can be used for.

Notice that the forked thread doesn't need to be passed any parameters, as is common in C threads packages. The forked action is a full closure that captures the values of its free variables. In this case, the forked action is `(serviceConn config conn)`, which obviously captures the free variables `config` and `conn`.

A thread may go to sleep for a specified number of microseconds by calling `threadDelay`:

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

`forkIO` is dangerous in a similar way that `unsafePerformIO` is dangerous (Section 2.6). I/O actions performed in the parent thread may interleave in an arbitrary fashion with I/O actions performed in the forked thread. Sometimes that is fine (e.g. the threads are painting different windows on the screen), but at other times we want the threads to co-operate more closely. To support such co-operation we need a synchronisation mechanism, which is what we discuss next.

4.2 Communication and `MVars`

Suppose we want to add some sort of throttling mechanism, so that when there are more than N threads running the server does something different (e.g. stops accepting new connections or something). To implement this we need to keep track of the total number of (active) forked threads. How can we do this? The obvious solution is to have a counter that the forked thread increments when it begins, and decrements when it is done. But we must of course be careful! If there are lots of threads all hitting on the same counter we must make sure that we don't get race hazards. The increments and decrements must be indivisible.

To this end, Concurrent Haskell supports a synchronised version of an `IORef` called an `MVar`:

```
data MVar a   -- Abstract
newEmptyMVar :: IO (MVar a)
takeMVar     :: MVar a -> IO a
putMVar      :: MVar a -> a -> IO ()
```

Like an `IORef`, an `MVar` is (a reference to) a mutable location that either can contain a value of type `a`,

or can instead be empty. Like `newIORef`, `newEmptyMVar` creates an `MVar` but, unlike an `IORef`, the `MVar` is created empty.

`putMVar` fills an empty `MVar` with a value, and `takeMVar` takes the contents of an `MVar` out, leaving it empty. If it was empty in the first place, the call to `takeMVar` blocks until another thread fills it by calling `putMVar`. A call to `putMVar` on an `MVar` that is already full blocks until the `MVar` becomes empty[7].

With the aid of `MVars` it is easy to implement our counter:

```
acceptConnections :: Config -> Socket -> IO ()
acceptConnections config socket
  = do { count <- newEmptyMVar ;
         putMVar count 0 ;
         forever (do { conn <- accept socket ;
                       forkIO (do { inc count ;
                                    serviceConn config conn ;
                                    dec count})
```

```
            }) }

    inc,dec :: MVar Int -> IO ()
    inc count = do { v <- takeMVar count; putMVar count (v+1) }
    dec count = do { v <- takeMVar count; putMVar count (v-1) }
```

Presumably there would also be some extra code in `acceptConnections` to inspect the value of the counter, and take some action if it gets too large.

The update of the counter, performed by `inc` and `dec` is indivisible because, during the brief moment while `inc` has read the counter but not yet written it back, the counter location is empty. So any other thread that tries to use `inc` or `dec` at that moment will simply block.

4.3 Semantics

One of the advantages of the operational semantics we set up in Section 3 is that it can readily be extended to support concurrency and `MVar`s. The necessary extensions are given in Figure 6:

- We add new values to represent (a) each new primitive `IO` operation; (b) the name of an `MVar` m, and a thread t; (c) the integer argument of a `threadDelay`, d.
- We extend program states by adding a form for an `MVar`, both in the full state $\langle M \rangle_m$, and in the empty state $\langle\rangle_m$; and a form for a named thread $\{M\}_t$.
- We provide transition rules for the new primitives.

Rules (FORK) and (NEWM) work in a very similar way as the (NEWIO) rule that we described in Section 3.5. In particular, they use ν in an identical fashion to control the new names that are required. Rules (PUTM) and (TAKEM) are similar to (WRITEIO) and (READIO), except that (TAKEM) leaves the `MVar` empty, while (PUTM) fills it.

For the first time, the semantics of the program has become non-deterministic. If there are two threads both of which want to take the contents of an `MVar`, the semantics leaves deliberately unspecified which one "wins". Once it has emptied the `MVar` with rule (TAKEM), however, the other thread can make no progress until some other thread fills it.

The rule (DELAY) deals with `threadDelay`. To express the delay, I have invented an extra event $\$d$, which means "$d$ microseconds elapse". Recall that an event indicates interaction with the external world

[7] This represents a change from an earlier version of Concurrent Haskell, in which `putMVar` on a full `MVar` was a program error.

$$
\begin{array}{rcl}
m & \in & \texttt{MVar} \\
t & \in & \texttt{ThreadId} \\
d & \in & \mathit{Integer} \\
\\
V & ::= & \ldots \texttt{forkIO}\, M \mid \texttt{threadDelay}\, d \mid t \mid d \\
  & \mid & \texttt{putMVar}\, m\, N \mid \texttt{takeMVar}\, m \mid \texttt{newEmptyMVar} \mid m
\end{array}
$$

$$
\begin{array}{rcll}
P, Q, R & ::= & \ldots & \\
 & \mid & \{M\}_t & \text{A thread called } t \\
 & \mid & \langle M \rangle_m & \text{An } \texttt{MVar} \text{ called } m \text{ containing } M \\
 & \mid & \langle\rangle_m & \text{An empty } \texttt{MVar} \text{ called } m
\end{array}
$$

$$
\frac{u \notin \mathit{fn}\,(M, \mathbb{E})}{\{\mathbb{E}[\texttt{forkIO}\, M]\}_t \;\rightarrow\; \nu u.(\{\mathbb{E}[\texttt{return}\, u]\}_t \mid \{M\}_u)} \;(\mathit{FORK})
$$

$$
\frac{m \notin \mathit{fn}\,(\mathbb{E})}{\{\mathbb{E}[\texttt{newEmptyMVar}]\}_t \;\rightarrow\; \nu m.(\{\mathbb{E}[\texttt{return}\, m]\}_t \mid \langle\rangle_m)} \;(\mathit{NEWM})
$$

$$
\begin{array}{rcll}
\{\mathbb{E}[\texttt{takeMVar}\, m]\}_t \mid \langle M \rangle_m & \rightarrow & \{\mathbb{E}[\texttt{return}\, M]\}_t \mid \langle\rangle_m & (\mathit{TAKEM}) \\
\{\mathbb{E}[\texttt{putMVar}\, m\, M]\}_t \mid \langle\rangle_m & \rightarrow & \{\mathbb{E}[\texttt{return ()}]\}_t \mid \langle M \rangle_m & (\mathit{PUTM}) \\
\\
\{\mathbb{E}[\texttt{threadDelay}\, d]\}_t & \xrightarrow{\$d} & \{\mathbb{E}[\texttt{return ()}]\}_t & (\mathit{DELAY})
\end{array}
$$

Figure 6: Extensions to support concurrency

(Section 3.3), so I am modelling a delay as an interaction with an external clock. This is not very satisfactory (e.g. I/O events are presumably queued up, but clock ticks should not be), but it gives the general idea.

Notice that there is no explicit rule for "blocking" a thread when it tries to take the contents of an `MVar` that is empty. All that happens is that there is no valid transition rule involving that thread, so it stays unchanged in the program state until the `MVar` it is trying to take is filled.

4.4 Channels

The thread created by `forkIO` and its parent thread can each independently perform input and output. We can think of the state of the world as a shared, mutable object, and race conditions can, of course, arise. For example, if two threads are foolish enough to write to the same file, say, bad things are likely to happen.

But what if we *want* to have two threads write to the same file, somehow merging their writes, at some suitable level of granularity? Precisely this behaviour is needed in our web server, because we want to log errors found by the client-service threads to a single error-log file. The simplest thing to do is to create a single thread whose business is to write to the error-log file; to log an error, a client-service thread need only send a message to the error-logging thread. But we have just pushed the problem to a different place: what does it mean to "send a message"?

Using `MVars` we can define a new type of buffered channels, which we will implement in this section:

```
type Channel a = ...given later...
newChan :: IO (Channel a)
putChan :: Channel a -> a -> IO ()
```

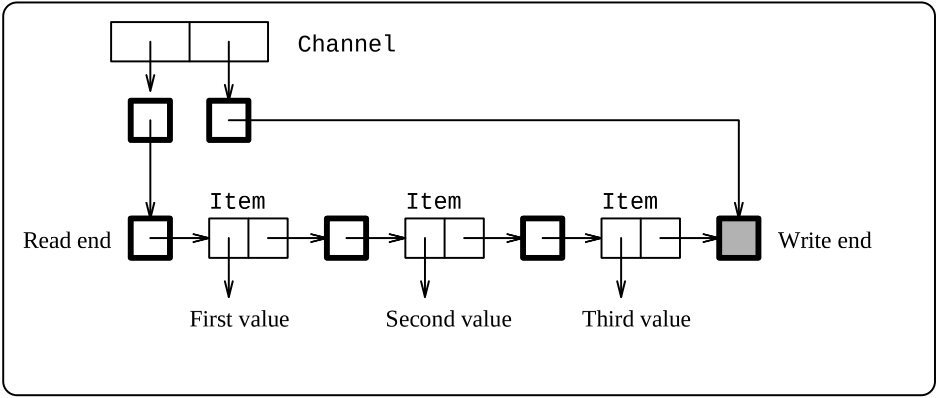


Figure 7: A channel with unbounded buffering

```
getChan :: Channel a -> IO a
```

A `Channel` permits multiple processes to write to it, and read from it, safely. The error-logging thread can now repeatedly do `getChan`, and write the value it receives into the file; meanwhile a client-service thread wanting to log an error can use `putChan` to send the error message to the error logger.

One possible implementation of `Channel` is illustrated in Figure 7. The channel is represented by a pair of `MVar`s (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 `MVar`s 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` (a data type we will define shortly):

```
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 = MkItem a (Stream a)
```

A `Stream` can therefore be thought of as a list, consisting of alternating `Item`s and full `MVar`s, 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 `MVar`s, plus one (empty) `MVar` for the stream itself:

```
newChan = do { read  <- newEmptyMVar ;
               write <- newEmptyMVar ;
               hole  <- newEmptyMVar ;
               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
  = do { new_hole <- newEmptyMVar ;
         old_hole <- takeMVar write ;
         putMVar write new_hole ;
         putMVar old_hole (MkItem val new_hole) }
```

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

```
getChan (read,write)
  = do { head_var <- takeMVar read ;
         MkItem val new_head <- takeMVar head_var ;
         putMVar read new_head ;
         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)
  = do { new_read <- newEmptyMVar ;
         hole <- readMVar write ;
         putMVar new_read hole ;
         return (new_read, write) }
```

To make the code clearer, I have used an auxiliary function, `readMVar`, which reads the value of an `MVar`, *but leaves it full*:

```
readMVar :: MVar a -> IO a
readMVar var = do { val <- takeMVar var ;
                    putMVar var val ;
                    return val }
```

But watch out! We need to modify `getChan` as well. In particular, we must change the call "`takeMVar head_var`" to "`readMVar head_var`". The MVars in the bottom row of Figure 7 are used to block the consumer when it catches up with the producer. If there are two consumers, it is essential that they can both march down the stream without intefering with each other. Concurrent programming is tricky!

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

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

4.5 Summary

Adding `forkIO` and `MVar`s to Haskell leads to a qualitative change in the sorts of applications one can write. The extensions are simple to describe, and our operational semantics was readily extended to de-


```
forkIO       ::  IO a -> IO ThreadId
threadDelay  ::  Int -> IO ()             -- Sleep for n microseconds

data MVar a      -- Abstract
newEmptyMVar ::  IO (MVar a)              -- Created empty
newMVar      ::  a -> IO (MVar a)         -- Initialised

takeMVar     ::  MVar a -> IO a           -- Blocking take
putMVar      ::  MVar a -> a -> IO ()     -- Blocking put

tryTakeMVar  ::  MVar a -> IO (Maybe a)   -- Non-blocking take
tryPutMVar   ::  MVar a -> a -> IO Bool   -- Non-blocking put
isEmptyMVar  ::  MVar a -> IO Bool        -- Test for emptiness
```

Figure 8: The most important concurrent operations

scribe them. Figure 8 lists the main operations in Concurrent Haskell, including some that we have not discussed.

You will probably have noticed the close similarity between `IORef`s (Section 2.5) and `MVar`s (Section 4.2). Are they both necessary? Probably not. In practice we find that we seldom use `IORef`s at all:

- Although they have slightly different semantics (an `IORef` cannot be empty) it is easy to simulate an `IORef` with an `MVar` (but not vice versa).
- An `MVar` is not much more expensive to implement than an `IORef`.
- An `IORef` is fundamentally unsafe in a concurrent program, unless you can prove that only one thread can access it at a time.

I introduced `IORef`s in these notes mainly as a presentational device; they allowed me to discuss the idea of updatable locations, and the operational machinery to support them, before getting into concurrency.

While the primitives are simple, they are undoubtedly primitive. `MVar`s are surprisingly often useful "as is", especially for holding shared state, but they are a rather low-level device. Nevertheless, they provide the raw material from which one can fashion more sophisticated abstractions, and a higher-order language like Haskell is well suited for such a purpose. `Channel`s are an example of such an abstraction, and we give several more in [35]. Einar Karlsen's thesis describes a very substantial application (a programming workbench) implemented in Concurrent Haskell, using numerous concurrency abstractions [22].

It is not the purpose of these notes to undertake a proper comparative survey of concurrent programming, but I cannot leave this section without mentioning two other well-developed approaches to concurrency in a declarative setting. Erlang is a (strict) functional language developed at Ericsson for programming telecommunications applications, for which purpose it has been extremely successful [3]. Erlang must be the most widely-used concurrent functional language in the world. Concurrent ML (CML) is a concurrent extension of ML, with a notion of first-class events and synchronisation constructs. CML's *events* are similar, in some ways, to Haskell's `IO` actions. CML lays particular emphasis on concurrency abstractions, and is well described in Reppy's excellent book [41].

5 Exceptions and timeouts

The next member of the Awkward Squad is robustness and error recovery. A robust program should not collapse if something unexpected happens. Of course, one tries to write programs in such a way that they will not fail, but this approach alone is insufficient. Firstly, programmers are fallible and, secondly, some failures simply cannot be avoided by careful programming.

Our web server, for example, should not cease to work if

- A file write fails because the disk is full.
- A client requests a seldom-used service, and that code takes the head of an empty list or divides by zero.
- A client vanishes, so the client-service thread should time out and log an error.
- An error in one thread makes it go into an infinite recursion and grow its stack without limit.

All these events are (hopefully) rare, but they are all unpredictable. In each case, though, we would like our web server to recover from the error, and continue to offer service to existing and new clients.

We cannot offer this level of robustness with the facilities we have described so far. We could check for failure on every file operation, though that would be rather tedious. We could try to avoid dividing by zero — but we will never know that we have found every bug. And timeouts and loops are entirely inaccessible.

This is, of course, exactly what exceptions were invented for. An exception handler can enclose an arbitrarily large body of code, and guarantee to give the programmer a chance to recover from errors arising anywhere in that code.

5.1 Exceptions in Haskell 98

Like many languages, Haskell's `IO` monad offers a simple form of exception handling. I/O operations may *raise* an exception if something goes wrong, and that exception can be *caught* by a handler. Here are the primitives that Haskell 98 offers:

```
userError :: String -> IOError
ioError   :: IOError -> IO a
catch     :: IO a -> (IOError -> IO a) -> IO a
```

You can raise an exception by calling `ioError` passing it an argument of type `IOError`. You can construct an `IOError` from a string using `userError`. Finally, you can catch an exception with `catch`. The call `(catch a h)` is an action that, when performed, attempts to perform the action `a` and return its results. However, if performing `a` raises an exception, then `a` is abandoned, and instead `(h e)` is returned, where `e` is the `IOError` in the exception.

$$
\begin{array}{rcl}
e & \in & \texttt{Exception} \\[1em]
V & ::= & \ldots \mid \texttt{ioError}\, e \mid \texttt{catch}\, M\ N \\[1em]
\mathbb{E} & ::= & [\cdot] \mid \mathbb{E}\, \texttt{>>=}\, M \mid \texttt{catch}\, \mathbb{E}\ M
\end{array}
$$

$$
\begin{array}{rcll}
\{\mathbb{E}[\texttt{ioError}\, e\, \texttt{>>=}\, M]\}_t & \rightarrow & \{\mathbb{E}[\texttt{ioError}\ e]\}_t & (\mathit{IOERROR}) \\
\{\mathbb{E}[\texttt{catch}\,(\texttt{ioError}\ e)\ M]\}_t & \rightarrow & \{\mathbb{E}[M\ e]\}_t & (\mathit{CATCH}1) \\
\{\mathbb{E}[\texttt{catch}\,(\texttt{return}\, N)\ M]\}_t & \rightarrow & \{\mathbb{E}[\texttt{return}\, N]\}_t & (\mathit{CATCH}2)
\end{array}
$$

Figure 9: Extensions for exceptions

Here is an example of how we might extend our main web-server loop:

```
acceptConnections :: Config -> Socket -> IO ()
acceptConnections config socket
  = forever (do { conn <- accept socket ;
                  forkIO (service conn) }
  where
    service :: Connection -> IO ()
    service conn = catch (serviceConn config conn)
                         (handler conn)

    handler :: Connection -> Exception -> IO ()
    handler conn e = do { logError config e ;
                          hClose (fst conn) }
```

Now the forked thread `(service conn)` has an exception handler wrapped around it, so that if anything goes wrong, `handler` will be invoked. This handler logs the error (presumably by sending a message to the error-logging thread through a channel held in `config`), and closes the connection handle `h`.

Figure 9 gives the extra semantics required to support Haskell 98 exceptions, in a style that by now will be familiar. The extra evaluation context says that we should evaluate inside a `catch`. Rule (IOERROR) says that a call to `ioError` is propagated by `(>>=)`; this is what corresponds to "popping the stack" in a typical implementation. Rules (CATCH1) describes what happens when the exception meets a `catch`: it is passed on to the handler. Lastly, (CATCH2) explains that `catch` does nothing if execution of the protected code terminates normally with `return` N.

The Haskell 98 design falls short in two ways:

- It does not handle things that might go wrong in purely-functional code, because an exception can only be raised in the `IO` monad. A pattern-match failure[8], or division by zero, brings the entire program to a halt. We address this problem in Section 5.2

- It does not deal with *asynchronous* exceptions. A synchronous exception arises as a direct result of executing some piece of code — opening a non-existent file, for example. Synchronous exceptions can be raised only at well-defined places. An asynchronous exception, in contrast, is raised by something in the thread's environment: a timeout or user interrupt is an asynchronous exception. It is useful to treat resource exhaustion, such as stack overflow, in the same way. An asynchronous exception can strike at any time, and this makes them much harder to deal with than their synchronous cousins. We tackle asynchronous exceptions in Section 5.3

[8] A pattern-match failure occurs when a function defined by pattern-matching is applied to a value for which no pattern matches. Example: taking the head of an empty list.

5.2 Synchronous exceptions in pure code

Why does Haskell 98 not allow the program to raise an exception in purely-functional code? The reason is that, as with input/output, Haskell's unconstrained order of evaluation makes it hard to say what the program means. Suppose we invented a new primitive to raise an exception:

```
throw :: Exception -> a
```

(`throw` differs from `ioError` in that it lacks an `IO` on its result type.) There are two difficulties:

(a) Consider the following expression:

```
length [throw ex1]
```

Does the expression raise exception `ex1`? Since `length` does not evaluate the elements of its argument list, the answer is presumably "no". So *whether an exception is raised depends on how much evaluation takes place*.

(b) Which exception does the following expression raise, `ex1` or `ex2`?

```
throw ex1 + throw ex2
```

The answer clearly depends on the order in which the arguments to `(+)` are evaluated. So *which exception is raised depends on evaluation order*.

As with input/output (right back in Section 1), one possibility is to fully define evaluation order and, as before, we reject that alternative.

5.2.1 Imprecise exceptions

The best approach is to take the hint from denotational semantics. The purely-functional part of the language should have a straightforward denotational semantics, and that requires us to answer the question: "what value does `throw e` return?". The answer must be "an exceptional value". So we divide the world of values (or denotations) into *ordinary values* (like `'a'` or `True` or `132`) and *exceptional values*. This is not a new idea. The IEEE Floating Point standard defines certain bit-patterns as "not-a-numbers", or NaNs. A NaN is returned by a floating point operation that fails in some way, such as division by zero. Intel's IA-64 architecture extends this idea to arbitrary data types, using "not-a-thing" (NaT) values to represent the result of speculative operations that have failed. In our terminology, a NaN or NaT is an exceptional value.

So `throw` simply constructs an exceptional value. It is a perfectly well-behaved value provided you never actually evaluate it; only then is the exception raised. The situation is very similar to that for a divergent (non-terminating) expression in a lazy language. Useful programs may contain such values; the program will only diverge if it actually evaluates the divergent term.

That deals with point (a) above, but how about (b)? A good solution is to say that the denotation of an expression is

- A single ordinary value, or
- A *set* of exceptions.

By making the denotation into a *set* of exceptions we can finesse the question of which exception is raised if many could be. Let us return to our troublesome example

```
throw ex1 + throw ex2
```


The denotation of this expression is now an exceptional value consisting of a set of two exceptions, `ex1` and `ex2`. In saying this, we do not need to say anything about evaluation order.

I am *not* suggesting that an implementation should actually *construct* the set of exceptions. The idea is that an implementation can use an entirely conventional exception-handling mechanism: when it evaluates an exceptional value, it rolls back the stack looking for a handler. In effect it chooses a single member of the set of exceptions to act as its representative [16].

5.2.2 Catching an imprecise exception

I describe this scheme as using "imprecise" exceptions, because we are deliberately imprecise about which exception is chosen as the representative. How, then, can we catch and handle an exception? At first we might try a non-`IO` version of `catch`:

```
bogusCatch :: a -> (Exception -> a) -> a      -- Bogus
```

`bogusCatch` evaluates its first argument; if it is an ordinary value, `bogusCatch` just returns it; if it is an exceptional value, `bogusCatch` applies the hander to the exception. But `bogusCatch` is problematic if the exceptional value contains a set of exceptions – which member of the set should be chosen? The trouble is that if the compiler decided to change evaluation order (e.g. optimisation is switched on) a different exception might be encountered, and the behaviour of the program would change.

A better approach is to separate the choice of which exception to throw from the exception-catching business:

```
evaluate :: a -> IO a
```

`evaluate x` evaluates its argument `x`; if the resulting value is an ordinary value, `evaluate` behaves just like `return`, and just returns the value. If `x` instead returns an exceptional value, `evaluate` chooses an arbitrary member, say `e`, from the set of exceptions, and then behaves just like `ioError e`; that is, it throws the exception `e`. So, for example, consider these four actions:

```
a1, a2, a3, a4 :: IO ()
a1 = do { x <- evaluate 4; print x }
a2 = do { evaluate (head []); print "no" }
a3 = do { return   (head []); print "yes" }
a4 = do { xs <- evaluate [1 ‘div‘ 0]; print (length xs) }
```

The first simply evaluates `4`, binds it to `x`, and prints it; we could equally well have written `(return 4)` instead. The second evaluates `(head [])`, finds an exceptional value, and throws an exception in the `IO` monad; the following `print` never executes. In contrast `a3` instead returns the exceptional value, ignores it, and prints `yes`. Lastly, `a4` evaluates the list ``[1 `div` 0]``, binds it to `xs`, takes its length, and prints the result. The list contains an exceptional value, but `evaluate` only evalutes the top level of its argument, and does not look inside its recursive structure (c.f. the `length` example in Section 5.2).

Now consider the case where the argument of `evaluate` is a set of exceptions; for example

```
evaluate (throw ex1 + throw ex2)
```

Since `evaluate x` is an I/O action (of type `IO t` if `x` has type `t`), there is no reason to suppose that it will choose the same member from the set of exceptions each time you run the program. It is free to perform input/output, so it can consult some external oracle (whether it is raining, say) to decide which member of the set to choose. More concretely, suppose we catch the exception like this:

```
catch (evaluate (throw ex1 + throw ex2)) h
```

(Recall that `catch` and its semantics was defined in Section 5.1.) The handler `h` will be applied to either `ex1` or `ex2`, and there is no way to tell which. It is up to `evaluate` to decide. This is different from `bogusCatch`, because the non-deterministic choice is made by an I/O action (`evaluate`) and not by a pure function (`bogusCatch`). I/O actions are not required to return the same result given the same input,

$$V \quad ::= \quad \ldots \mid \texttt{evaluate}\, M$$

$$\frac{\mathcal{E}[\![M]\!] = Ok\ V}{\{\mathbb{E}[\texttt{evaluate}\, M]\}_t \;\rightarrow\; \{\mathbb{E}[\texttt{return}\, V]\}_t}\ (\mathit{EVAL1})$$

$$\frac{\mathcal{E}[\![M]\!] = Bad\ S \quad e \in S}{\{\mathbb{E}[\texttt{evaluate}\, M]\}_t \;\rightarrow\; \{\mathbb{E}[\texttt{ioError}\, e]\}_t}\ (\mathit{EVAL2})$$

$$\frac{\mathcal{E}[\![M]\!] = Ok\ V \quad M \not\equiv V}{\{\mathbb{E}[M]\}_t \;\rightarrow\; \{\mathbb{E}[V]\}_t}\ (\mathit{FUN1})$$

$$\frac{\mathcal{E}[\![M]\!] = Bad\ S \quad e \in S}{\{\mathbb{E}[M]\}_t \;\rightarrow\; \{\mathbb{E}[\texttt{ioError}\, e]\}_t}\ (\mathit{FUN2})$$

$$\frac{M \neq (N_1 \texttt{>>=} N_2) \qquad M \neq (\texttt{catch}\, N_1\ N_2)}{\{\mathbb{E}_1[\texttt{throwTo}\, t\ e]\}_s \mid \{\mathbb{E}_2[M]\}_t \;\rightarrow\; \{\mathbb{E}_1[\texttt{return ()}]\}_s \mid \{\mathbb{E}_2[\texttt{ioError}\, e]\}_t}\ (\mathit{INT})$$

Figure 10: Further extensions for exceptions

whereas pure functions are. In practice, `evaluate` will not *really* be non-deterministic; the decision is really taken by the evaluation order chosen by the compiler when it compiles the argument to `evaluate`.

Notice what we have done:

- An exception can be *raised* anywhere, including in purely-functional code. This is tremendously useful. For example, pattern-match failure can now raise an exception rather than bringing execution to a halt. Similarly, Haskell 98 provides a function `error`:

  ```
  error :: String -> a
  ```

  When `error` is called, the string is printed, and execution comes to a halt. In our extended version of Haskell, `error` instead raises an exception, which gives the rest of the program a chance to recover from the failure.

- An exception can only be *caught* by `catch`, which is in the `IO` monad. This confines recovery to the monadic-I/O layer of the program, unlike ML (say) where you can catch an exception anywhere. In my view, this restriction is not irksome, and has great semantic benefits. In particular, *by confining the non-deterministic choice to the* `IO` *monad we have prevented non-determinism from infecting the entire language*.

5.2.3 Semantics of imprecise exceptions

This approach to synchronous exceptions in Haskell is described in much more detail in [37]. In particular, the paper describes how to extend a standard denotational semantics to include exceptional values, something we have not treated formally here. We will not discuss that here, for lack of space, but will content ourselves with saying that the meaning function $\mathcal{E}[\![M]\!]$ returns either $Ok\ v$ for an ordinary value v, or $Bad\ S$ for an exceptional value, where S is a non-empty set of exceptions. For example, here is the semantics of addition:

$$\mathcal{E}[\![e_1\texttt{+}e_2]\!] = \mathcal{E}[\![e_1]\!]\ +'\ \mathcal{E}[\![e_2]\!]$$

where $+'$ is an addition function defined over the semantic domain of values, thus:

$$\begin{array}{rcl} (Ok\ v_1)\ +'\ (Ok\ v_2) & = & Ok\ (v_1 + v_2) \\ (Ok\ v_1)\ +'\ (Bad\ s_2) & = & Bad\ s_2 \\ (Bad\ s_1)\ +'\ (Ok\ v_2) & = & Bad\ s_1 \\ (Bad\ s_1)\ +'\ (Bad\ s_2) & = & Bad\ (s_1 \cup s_2) \end{array}$$

The first equation deals with the normal case. The second and third deal with the case when one or other of the arguments throws an exception. The last equation handles the case when both arguments throw an exception; in this case $+'$ takes the union of the exceptions that can be thrown by the two arguments. The whole point is that $+'$ is commutative, so that $\mathcal{E}[\![e_1 \texttt{+} e_2]\!] = \mathcal{E}[\![e_2 \texttt{+} e_1]\!]$.

Given this, Figure 10 gives the extra semantics for `evaluate`. If the argument to `evaluate` is an ordinary value, `evaluate` just returns that value (EVAL1); if the value is an exceptional value, `evaluate` chooses an arbitrary member of the set of exceptions, and throws that exception using `ioError`. This deliberately-unconstrained choice is where the non-determinism shows up in the operational semantics.

Since $\mathcal{E}[\![]\!]$ has changed we must do something to rule (FUN). This is a place where our semantics forces us to recognise something we might otherwise have forgotten. Rules (FUN1) and (FUN2) replace (FUN). (FUN2) says that if the next action to perform is itself an exceptional value, then we should just propagate that as an `IO`-monad exception using `ioError`. If it is not, then we behave just like (FUN). Here is an example that shows the importance of this change:

```
catch (if (1/0) then a1 else a2) recovery_code
```

Before `catch` can perform the action that is its first argument, it must evaluate it; in this case, evaluating it gives divide-by-zero exception, and rule (FUN2) propagates that into an `ioError`.

The `Exception` data type is really the same as `IOError`, except that "`IOError`" does not seem an appropriate name any more. To keep things simple, we just say that `IOError` is a synonym for `Exception`. To summarise, we now have the following primitives:

```
type IOError = Exception
throw    :: Exception -> a
evaluate :: a -> IO a
ioError  :: IOError -> IO a
catch    :: IO a -> (Exception -> IO a) -> IO a
```

5.3 Asynchronous exceptions

We now turn our attention to asynchronous exceptions. For asynchronous exceptions, we add the following new primitive:

```
throwTo :: ThreadId -> Exception -> IO ()
```

This allows one thread to interrupt another. So far as the interrupted thread is concerned, the situation is just as if it abruptly called `ioError`; an exception is raised and propagated to the innermost enclosing `catch`. This is where the `ThreadId` of a forked thread becomes really useful: we can use it as a handle to send an interrupt to another thread. One thread can raise an exception in another only if it has the latter's `ThreadId`, which is returned by `forkIO`. So a thread is in danger of interruption only from its parent, unless its parent passes on its `ThreadId` to some other thread.

5.3.1 Using asynchronous exceptions

Using `throwTo` we can implement a variety of abstractions that are otherwise inaccessible. For example, we can program the combinator `parIO`, which "races" its two argument actions against each other in parallel. As soon as one terminates, it kills the other, and the overall result is the one returned by the "winner".


```
parIO :: IO a -> IO a -> IO a
```

How can we implement this? We can use an MVar to contain the overall result. We spawn two threads, that race to fill the result MVar; the first will succeed, while the second will block. The parent takes the result from the MVar, and then kills both children:

```
parIO :: IO a -> IO a -> IO a
parIO a1 a2
  = do { m <- newEmptyMVar ;
         c1 <- forkIO (child m a1) ;
         c2 <- forkIO (child m a2) ;
         r <- takeMVar m ;
         throwTo c1 Kill ;
         throwTo c2 Kill ;
         return r
    }
  where
    child m a = do { r <- a ; putMVar m r }
```

Using `parIO` we can implement a simple timeout:

```
timeout :: Int -> IO a -> IO (Maybe a)
```

The idea here is that `(timeout n a)` returns `Nothing` if `a` takes longer than `n` microseconds to complete, and `Just r` otherwise, where `r` is the value returned by `a`:

```
timeout :: Int -> IO a -> IO (Maybe a)
```

```
timeout n a = parIO (do { r <- a; return (Just r) })
                    (do { threadDelay n; return Nothing })
```

Now we might want to answer questions like this: what happens if a thread is interrupted (via a `throwTo`) while it is executing under a timeout? We can't say for sure until we give a semantics to `throwTo`, which is what we do next.

5.3.2 Semantics of asynchronous exceptions

We can express the behaviour of `throwTo` nicely in our semantics: a `throwTo` in one thread makes the target thread abandon its current action and replace it with `ioError`:

$$\frac{M \neq (N_1 \texttt{>>=} N_2) \qquad M \neq (\texttt{catch}\, N_1\; N_2)}{\{\mathbb{E}_1[\texttt{throwTo}\, t\; e]\}_s \mid \{\mathbb{E}_2[M]\}_t \;\rightarrow\; \{\mathbb{E}_1[\texttt{return ()}]\}_s \mid \{\mathbb{E}_2[\texttt{ioError}\, e]\}_t} \; (INT)$$

("(INT)" is short for "interrupt".) The conditions above the line are essential to ensure that the context $\mathbb{E}_2$ is maximal; that is, it includes all the active `catch`es.

It should be clear that external interrupts, such as the user pressing Control-C, can also be modeled in this way. Before we can write the semantics we have to answer several questions. Does a Control-C interrupt every thread, or just a designated thread? If the latter, how does a thread get designated? These are good questions to be forced to answer, because they really do make a difference to the programmer.

Having a semantics is very helpful in answering questions like: what happens if a thread is interrupted when it is blocked waiting for an `MVar`? In the semantics, such a thread is simply stuck, with a `takeMVar` at the active site, so (INT) will cause the `takeMVar` to be replaced with `ioError`. So being blocked on an `MVar` doesn't stop a thread receiving an interrupt.

Now we can say what happens to a thread that executes a sub-computation using `timeout`, but is interrupted by `throwTo` while it is waiting for the sub-computation to complete. The parent thread receives the interrupt while it is blocked on the "`takeMVar m`" inside `parIO` (Section 5.3.1); so it abandons the wait and proceeds to the innermost `catch` handler. But that means that the two threads spawned by `parIO`

are not killed, and we probably want them to be. So we have to go back to fix up `parIO` somehow. In fact this turns out to be tricky to do: we have to make sure that there is no "window" in which the parent has spawned a child thread but has not set up a handler that will kill the child if the parent is interrupted.

Indeed, programming in the presence of asynchronous exceptions is notoriously difficult, so much so that Modula-3, for example, simply outlaws them. (Instead, well-behaved threads regularly poll an *alert* flag, and commit suicide if it is set [33].) Haskell differs from Modula in two ways that are relevant here. First, there are fewer side effects, so there are fewer windows of vulnerability to worry about. Second, there are large parts of purely-functional code that we would like to be able to interrupt — and can indeed do so safely — but where any polling mechanism would be very undesirable. These considerations led us to define new primitive combinators to allow a thread to mask and un-mask external interrupts. This further complicates the semantics, but as a result we can write code where we have a chance of *proving* that it has no race hazards. The details are in [29].

5.4 Summary

This section on exceptions is the most experimental of our main themes. Two papers, [37, 29], give a great deal more detail on the design, which I have introduced here only in outline. Indeed, some aspects of the asynchronous-exceptions design are still in flux at the time of writing.

Adding exceptions undoubtedly complicates the language and its semantics, and that is never desirable. But they allow a qualitative change in the robustness of a program. Now, if there is a pattern match failure almost anywhere in the code of the web server, the system can recover cleanly. Without exceptions, such a failure would be fatal.

6 Interfacing to other programs

In the programming-language world, one rule of survival is simple: dance or die. It is not enough to make a beautiful language. You must also make it easy for programs written in your beautiful language to interact with programs written in other languages. Java, C++, and C all have huge, and hugely useful, libraries available. For example, our web server makes extensive use of socket I/O libraries written in C. It is fruitless to reproduce many of these libraries in Haskell; instead, we want to make it easy to call them. Similarly, if we want to plug a small Haskell program into a large project, it is necessary to enable other programs to call Haskell. It is hubristic to expect the Haskell part to always be "on top".

Haskell 98 does not specify any way to call foreign-language procedures, but there has been a lot of progress on this front in the last few years, which I survey in this section. In particular, a proposal has emerged for a Haskell language extension to support foreign-language interfacing. We will call this proposal *the Haskell Foreign Function Interface (FFI) proposal*; it is documented at `http://haskell.org/definition/ffi`.

6.1 Calling C from Haskell, and Haskell from C

Here is how you can call a C procedure from Haskell, under the FFI proposal:

```
foreign import ccall putChar :: Char -> IO ()
```

The `foreign` declaration brings into scope a Haskell function `putChar` with the specified type. When this function is called, the effect is to call a C procedure, also called `putChar`. Simple, eh? The complete syntax is given in Figure 11. The following points are worth noting:

- As usual, we use the `IO` monad in the result type of `putChar` to indicate that `putChar` may perform I/O, or have some other side effect. However, some foreign procedures may have purely-


```
decl       ::= foreign import callconv [safety] imp_entity varid :: ftype
             | foreign export callconv [safety] exp_entity varid :: ftype

callconv   ::= ccall | stdcall | ...other calling conventions...
safety     ::= safe | unsafe
imp_entity ::= [string]
exp_entity ::= [string]

ftype      ::= () | IO fatype | fatype | fatype -> ftype

fatype     ::= Int | Float | Double | Char | Bool
             | Ptr type | FunPtr type | StablePtr type
             | Int8 | Int16 | Int32 | Int64
             | Word8 | Word16 | Word32 | Word64
             | A Haskell newtype of a fatype
             | A Haskell type synonym for a fatype
```

Figure 11: The Haskell FFI proposal syntax

functional semantics. For example, the C `sin` function really is a function: it has no side effects. In this case it is extremely tiresome to force it to be in the `IO` monad. So the Haskell FFI allows one to omit the "`IO`" from the return type, thus:

```
foreign import ccall sin :: Float -> Float
```

The non-`IO` type indicates that the programmer takes on a proof obligation, in this case that foreign procedure is genuinely functional.

- The keyword "`ccall`" indicates the calling convention to use; that is, which arguments are passed in which registers, which on the stack, where the result is returned, and so on. The only other currently-defined calling convention at the moment is "`stdcall`", used on Win32 platforms.
- If the foreign procedure does not have the same name as its Haskell counterpart — for example, it might start with a capital letter, which is illegal for Haskell functions — you can specify the foreign name directly:

```
foreign import ccall "PutChar" putChar :: Char -> IO ()
```

- Foreign procedures may take several arguments. Their Haskell type is curried, as is usually the case for multi-argument Haskell functions, but on the C side the arguments are passed all at once, as is usual for C:

```
foreign import ccall drawLine :: Int -> Int -> IO ()
```

- There is a strictly limited range of Haskell types that can be used in arguments and results, namely the "atomic" types such as `Int`, `Float`, `Double`, and so on. So how can we pass structured types, such as strings or arrays? We address this question in Section 6.3.
- An implementation of the FFI proposal must provide a collection of new atomic types (Figure 11).

In particular, `Ptr` t is the type of uninterpreted[9] machine addresses; for example, a pointer to a `malloc`'d structure, or to a C procedure. The type t is a "phantom type", which allows the Haskell programmer to enforce the distinction between (say) the types `Ptr Foo` and `Ptr Baz`. No actual *values* of type `Foo` or `Baz` are involved.

[9] "Uninterpreted" in the sense that they are treated simply as bit patterns. The Haskell garbage collector does not follow the pointer.

"`foreign import`" lets you call a C procedure from Haskell. Dually, "`foreign export`" lets you expose a Haskell function as a C procedure. For example:

```
foreign export ccall "Foo" foo :: Int -> Int
foreign export ccall       bar :: Float -> IO Float
```

These declarations are only valid if the same module defines (or imports) Haskell functions `foo` and `bar`, which have the specified types. An exported function may have an `IO` type, but it does not have to — here, `bar` does, and `foo` does not. When the module is compiled, it will expose two procedures, `Foo` and `bar`, which can be called from C.

6.2 Dynamic calls

It is quite common to make an *indirect* call to an external procedure; that is, one is supplied with the address of the procedure and one wants to call it. An example is the dynamic dispatch of a method call in an object-oriented system, indirecting through the method table of the object.

To make such an indirect call from Haskell, use the `dynamic` keyword:

```
foreign import ccall "dynamic"
      foo :: FunPtr (Int -> IO Int) -> Int -> IO Int
```

The first argument must be of type `FunPtr` t, and is taken to be the machine address of the external procedure to be called. As in the case of `Ptr` t, the type t is used simply to express the distinction between pointers to procedures of different types.

There is also a way to export a dynamic Haskell value:

```
foreign import ccall "wrapper"
        mkCB :: (Int -> IO Int) -> IO (FunPtr (Int -> IO Int))
```

This declaration defines a Haskell function `mkCB`. When `mkCB` is given an arbitrary Haskell function of type `(Int->IO Int)`, it returns a C function pointer (of type `FunPtr (Int -> IO Int)`) that can be called by C. Typically, this `FunPtr` is then somehow passed to the C program, which subsequently uses it to call the Haskell function using a C indirect call.

6.3 Marshalling

Transferring control is, in some ways, the easy bit. Transferring data "across the border" is much harder. For "atomic" types, such as `Int` and `Float`, it is clear what to do, but for structured types, matters are much murkier.

For example, suppose we wanted to import a function that operates on strings:

```
foreign import ccall uppercase :: String -> String
```

- First there is the question of data representation. One has to decide either to alter the Haskell language implementation, so that its string representation is identical to that of C, or to translate the string from one representation to another at run time. This translation is conventionally called *marshalling*.

  Since Haskell is lazy, the second approach is required. In any case, it is tremendously constraining to try to keep common representations between two languages. For example, C terminates strings with a null character, but other languages may keep a length field. Marshalling, while expensive, serves to separate the implementation concerns of the two different languages.

- Next come questions of allocation and lifetime. Where should we put the translated string? In a static piece of storage? (But how large a block should we allocate? Is it safe to re-use the same block on the next call?) Or in Haskell's heap? (But what if the called procedure does something that triggers

garbage collection, and the transformed string is moved? Can the called procedure hold on to the string after it returns?) Or in C's `malloc`'d heap? (But how will it get deallocated? And `malloc` is expensive.)

- C procedures often accept pointer parameters (such as strings) that can be `NULL`. How is that to be reflected on the Haskell side of the interface? For example, if `uppercase` did something sensible when called with a `NULL` string (e.g. returns a `NULL` string) we might like the Haskell type for `uppercase` to be

```
foreign import ccall uppercase :: Maybe String -> Maybe String
```

  so that we can model `NULL` by `Nothing`.

The bottom line is this: there are many somewhat-arbitrary choices to make when marshalling parameters from Haskell to C and vice versa. And that's only C! There are even more choices when we consider arbitrary other languages.

What are we to do? The consensus in the Haskell community is this:

> We define a *language extension* that is as small as possible, and build *separate tools* to generate marshalling code.

The `foreign import` and `foreign export` declarations constitute the language extension. They embody just the part of foreign-language calls that cannot be done in Haskell itself, *and no more*. For example, suppose you want to import a procedure that draws a line, whose C prototype might look like this:

```
void DrawLine( float x1, float y1, float x2, float y2 )
```

One might ideally like to import this procedure with the following Haskell signature.

```
type Point = (Float,Float)
drawLine :: Point -> Point -> IO ()
```

The FFI proposal does not let you do this directly. Instead you have to do some marshalling yourself (in this case, unpacking the pairs):

```
type Point = (Float,Float)

drawLine :: Point -> Point -> IO ()
drawLine (x1,y1) (x2,y2) = dl_help x1 y1 x2 y2

foreign import ccall "DrawLine"
      dl_help :: Float -> Float -> Float -> Float -> IO ()
```

Writing all this marshalling code can get tedious, especially when one adds arrays, enumerations, in-out parameters passed by reference, NULL pointers, and so on. There are now several tools available that take some specification of the interface as input, and spit out Haskell code as output. Notably:

Green Card [34] is a pre-processor for Haskell that reads directives embedded in a Haskell module and replaces these directives with marshalling code. Using Green Card one could write

```
type Point = (Float,Float)
drawLine :: Point -> Point -> IO ()
%call (float x1, float y1) (float x2, float y2)
%code DrawLine( x1, y1, x2, y2 )
```

Green Card is C-specific, and doesn't handle the foreign-export side of things at all.

`C->Haskell` [8] reads both a Haskell module with special directives (or "binding hooks") and a standard C header file, and emits new Haskell module with all the marshalling code added. The advantage compared to Green Card is that less information need be specified in the binding hooks than in Green Card directives.

H/Direct [10] instead reads a description of the interface written in *Interface Definition Language* (IDL), and emits a Haskell module containing the marshalling code. IDL is a huge and hairy language, but it is neither Haskell-specific nor C-specific. H/Direct deals with both import and export, can read Java class files as well as IDL files, and can generate code to interface to C, COM, and Java.

It is well beyond the scope of these notes to give a detailed introduction to any of these tools here. However, in all cases the key point is the same: *any of these tools can be used with any Haskell compiler that implements the* `foreign` *declaration*. The very fact that there are three tools stresses the range of possible design choices, and hence the benefit of a clear separation.

6.4 Memory management

One of the major complications involved in multi-language programs is memory management. In the context of the Haskell FFI, there are two main issues:

Foreign objects. Many C procedures return a pointer or "handle", and expect the client to *finalise* it when it is no longer useful. For example: opening a file returns a file handle that should later be closed; creating a bitmap may allocate some memory that should later be freed; in a graphical user interface, opening a new window, or a new font, returns a handle that should later be closed. In each case, resources are allocated (memory, file descriptors, window descriptors) that can only be released when the client explicitly says so. The term *finalisation* is used to describe the steps that must be carried out when the resource is no longer required.

The problem is this: if such a procedure is imported into a Haskell program, how do we know when to finalise the handle returned by the procedure?

Stable pointers. Dually, we may want to pass a Haskell value into the C world, either by passing it as a parameter to a `foreign import`, or by returning it as a result of a `foreign export`. Here, the danger is not that the value will live too long, but that it will die too soon: how does the Haskell garbage collector know that the value is still needed? Furthermore, even if it does know, the garbage collector might move live objects around, which would be a disaster if the address of the old location of the object is squirreled away in a C data structure.

In this section we briefly survey solutions to these difficulties.

6.4.1 Foreign objects

One "solution" to the finalisation problem is simply to require the Haskell programmer to call the appropriate finalisation procedure, just as you would in C. This is fine, if tiresome, for I/O procedures, but unacceptable for foreign libraries that have purely functional semantics.

For example, we once encountered an application that used a C library to manipulate bit-maps [39]. It offered operations such as filtering, thresholding, and combining; for example, to 'and' two bit-maps together, one used the C procedure `and_bmp`:

```
bitmap *and_bmp( bitmap *b1, bitmap *b2 )
```

Here, `and_bmp` allocates a new bit-map to contain the combined image, leaving `b1` and `b2` unaffected. We can import `and_bmp` into Haskell like this:


```
data Bitmap = Bitmap      -- A phantom type
foreign import ccall
      and_bmp :: Ptr Bitmap -> Ptr Bitmap -> IO (Ptr Bitmap)
```

Notice the way we use the fresh Haskell type `Bitmap` to help ensure that we only give to `and_bmp` an address that is the address of a bitmap.

The difficulty is that there is no way to know when we have finished with a particular bit-map. The result of a call to `and_bmp` might, for example, be stored in a Haskell data structure for later use. The only time we can be sure that a bitmap is no longer needed is when the Haskell garbage collector finds that its `Ptr` is no longer reachable.

Rather than ask the garbage collector to track all `Ptr`s, we wrap up the `Ptr` in a *foreign pointer*, thus:

```
newForeignPtr :: Ptr a -> IO () -> IO (ForeignPtr a)
```

`newForeignPtr` takes a C-world address, and a finalisation action, and returns a `ForeignPtr`. When the garbage collector discovers that this `ForeignPtr` is no longer accessible, it runs the finalisation action.

To unwrap a foreign pointer we use `withForeignPtr`:

```
withForeignPtr :: ForeignPtr a -> (Ptr a -> IO b) -> IO b
```

(We can't simply unwrap it with a function of type `ForeignPtr a -> IO Ptr a` because then the foreign pointer itself might be unreferenced after the unwrapping call, and its finaliser might therefore be called before we are done with the `Ptr`.)

So now we can import `add_bmp` like this:

```
foreign import ccall "and_bmp"
      and_bmp_help :: Ptr Bitmap -> Ptr Bitmap -> IO (Ptr Bitmap)

foreign import ccall free_bmp :: Ptr Bitmap -> IO ()

and_bmp :: ForeignPtr Bitmap -> ForeignPtr Bitmap -> IO (ForeignPtr Bitmap)
and_bmp b1 b2 = withForeignPtr b1         (\ p1 ->
                withForeignPtr b2         (\ p2 ->
                do { r <- and_bmp_help p1 p2
                     newForeignObj r (free_bmp r) }))
```

The function `and_bmp` unwraps its argument `ForeignPtrs`, calls `and_bmp_help` to get the work done, and wraps the result back up in a `ForeignPtr`.

6.4.2 Stable pointers

If one wants to write a Haskell library that can be called by a C program, then the situation is reversed compared to foreign objects. The Haskell library may construct Haskell values and return them to the C caller. There is not much the C program can do with them directly (since their representation depends on the Haskell implementation), but it may manipulate them using other Haskell functions exported by the library.

As we mentioned earlier, we cannot simply return a pointer into the Haskell heap, for two reasons:

- The Haskell garbage collector would not know when the object is no longer required. Indeed, if the C program holds the *only* pointer to the object, the collector is likely to treat the object as garbage, because it has no way to know what Haskell pointers are held by the C program.

- The Haskell garbage collector may move objects around (GHC's collector certainly does), so the address of the object is not a stable way to refer to the object.

The straightforward, if brutal, solution to both of these problems is to provide a way to convert a Haskell value into a *stable pointer*:

```
newStablePtr   :: a -> IO (StablePtr a)
deRefStablePtr :: StablePtr a -> IO a
freeStablePtr  :: StablePtr a -> IO ()
```

The function `newStablePtr` takes an arbitrary Haskell value and turns it into a stable pointer, which has two key properties:

- First, it is stable; that is, it is unaffected by garbage collection. A `StablePtr` can be passed to C as a parameter or result to a `foreign import` or a `foreign export`. From the C side, a `StablePtr` looks like an `int`. The C program can subsequently pass the stable pointer to a Haskell function, which can get at the original value using `deRefStablePtr`.

- Second, calling `newStablePtr v` registers the Haskell value as a garbage-collection root, by installing a pointer to `v` in the *Stable Pointer Table* (SPT). Once you have called `newStablePtr v`, the value `v` will be kept alive indefinitely by the SPT, even if `v`, or even the `StablePtr` itself are no longer reachable.

  How, then, can `v` ever die? By calling `freeStablePtr`: This removes the entry from the SPT, so `v` can now die unless it is referenced some other way.

Incidentally, the alert reader may have noticed that `foreign import "wrapper"`, described in Section 6.2, must use stable pointers. Taking the example in that section, `mkCB` turns a Haskell function value into a plain `Addr`, the address of a C-callable procedure. It follows that `mkCB f` must register `f` as a stable pointer so that the code pointed to by the `Addr` (which the garbage collector does not follow) can refer to it. Wait a minute! How can we free the stable pointer that is embedded inside that `Addr`? You have to use this function:

```
freeHaskellFunctionPtr :: Addr -> IO ()
```

6.5 Implementation notes

It is relatively easy to implement the `foreign import` declaration. The code generator needs to be taught how to generate code for a call, using appropriate calling conventions, marshalling parameters from the small, fixed range of types required by the FFI. The `dynamic` variant of `foreign import` is no harder.

A major implementation benefit is that all the I/O libraries can be built on top of such `foreign import`s; there is no need for the code generator to treat `getChar`, say, as a primitive.

Matters are not much harder for `foreign export`; here, the code generator must produce a procedure that can be called by the foreign language, again marshalling parameters appropriately. `foreign import "wrapper"` is tricker, though, because we have to generate a single, static address that encapsulates a full Haskell closure. The only way to do this is to emit a little machine code at run-time; more details are given in [11][10].

6.6 Summary and related work

So far I have concentrated exclusively on interfacing to programs written in C. Good progress has also been made for other languages and software architectures:

[10] In that paper, `foreign import "wrapper"` is called "`foreign export dynamic`"; the nomenclature has changed slightly.

COM is Microsoft's Component Object Model, a language-independent, binary interface for composing software components. Because of its language independence COM is a very attractive target for Haskell. H/Direct directly supports both calling COM objects from Haskell, and implementing COM objects in Haskell [36, 10, 11, 26].

CORBA addresses similar goals to COM, but with a very different balance of design choices. H/Direct can read CORBA's flavour of IDL, but cannot yet generate the relevant marshalling and glue code. There is a good CORBA interface for the functional/logic language Mercury, well described in [20].

Lambada [30] offers a collection of Haskell libraries that makes it easy to write marshalling code for calling, and being called by, Java programs. Lambada also offers a tool that reads Java class files and emits IDL that can then be fed into H/Direct to generate the marshalling code. There is ongoing work on extending the `foreign` declaration construct to support Java calling conventions.

The actual Haskell FFI differs slightly from the one give here; in particular, there are many operations over the types `Addr`, `ForeignObj` and `StablePtr` that I have omitted. Indeed, some of the details are still in flux.

Finalisation can be very useful even if you are not doing mixed language working, and many languages support it, including Java, Dylan, Python, Scheme, and many others. Hayes gives a useful survey [13], while a workshop paper gives more details about the Glasgow Haskell Compiler's design for finalisers [28].

This section is notably less thorough and precise than earlier sections. I have given a flavour of the issues and how they can be tackled, rather than a detailed treatment. The plain fact is that interfacing to foreign languages is a thoroughly hairy enterprise, and no matter how hard we work to build nice abstractions, the practicalities are undoubtedly complicated. There are many details to be taken care of; important aspects differ from operating system to operating system; there are a variety of interface definition languages (C header files, IDL, Java class files etc); you have to use a variety of tools; and the whole area is moving quickly (e.g. the recent announcement of Microsoft's .NET architecture).

7 Have we lost the plot?

Now that we have discussed the monadic approach in some detail, you may well be asking the following question: once we have added imperative-looking input/output, concurrency, shared variables, and exceptions, have we not simply re-invented good old procedural programming? Have we "lost the plot" — that is, forgotten what the original goals of functional programming were?

I believe not. The differences between conventional procedural programming and the monadic functional style remain substantial:

- There is a clear distinction, enforced by the type system, between *actions* which may have side effects, and *functions* which may not. The distinction is well worth making from a software engineering point of view. A function can be understood as an independent entity. It can only affect its caller through the result it returns. Whenever it is called with the same arguments it will deliver the same result. And so on.

  In contrast, the interaction of an action with its caller is complex. It may read or write `MVar`s, block, raise exceptions, fork new threads... and none of these things are explicit in its type.

- No reasoning laws are lost when monadic I/O is added. For example, it remains unconditionally true that

$$\texttt{let}\ x = e\ \texttt{in}\ b \quad = \quad b[e/x]$$

There are no side conditions, such as "e must not have side effects". (There is an important caveat, though: I am confident that this claim is true, but I have not proved it.)

- In our admittedly-limited experience, most Haskell programs consist almost entirely of functions, not actions: a small monadic-I/O "skin" surrounds a large body of purely-functional code. While it is certainly possible to write a Haskell program that consists almost entirely of I/O, it is unusual to do so.

- Actions are first class values. They can be passed as arguments to functions, returned as results, stored in data structures, and so on. This gives unusual flexibility to the programmer.

Another good question is this: is the `IO` monad a sort of "sin-bin", used whenever we want to do something that breaks the purely-functional paradigm? Could we be a bit more refined about it? In particular, if we argue that it is good to know from the type of an expression that it has no side effects, would it not also be useful to express in the type some limits on the side effects it may cause? Could we have a variant of `IO` that allowed exceptions but not I/O? Or I/O but not concurrency? The answer is technically, yes of course. There is a long history of research into so-called *effect systems*, that track what kind of effects an expression can have [21]. Such effect systems can be expressed in a monadic way, or married with a monadic type system [51]. However, the overhead on the programmer becomes greater, and I do not know of any language that uses such a system[11]. An interesting challenge remains, to devise a more refined system that is still practical; there is some promising work in this direction [6, 51, 45, 5]. Meanwhile I argue that a simple pure-or-impure distinction offers an excellent cost/benefit tradeoff.

8 Summary

We have surveyed Haskell's monadic I/O system, along with three significant language extensions[12]. It is easy to extend a language, though! Are these extensions any good? Are they just an *ad hoc* set of responses to an *ad hoc* set of demands? Will every new demand lead to a new extension? Could the same effect be achieved with something simpler and more elegant?

I shall have to leave these judgments to you, gentle reader. These notes constitute a status report on developments in the Haskell community at the time of writing. The extensions I have described cover the needs of a large class of applications, so I believe we have reached at least a plateau in the landscape. Nevertheless the resulting language is undeniably complicated, and the monadic parts have a very imperative feel. I would be delighted to find a way to make it simpler and more declarative.

The extensions are certainly practical — everything I describe is implemented in the Glasgow Haskell compiler — and have been used to build real applications.

You can find a great deal of information about Haskell on the Web, at

`http://haskell.org`

There you will find the language definition, tutorial material, book reviews, pointers to free implementations, details of mailing lists, and more besides.

Acknowledgements

These notes have been improved immeasurably by many conversations with Tony Hoare. Thank you Tony! I also want to thank Peter Aachten, Ken Anderson, Richard Bird, Paul Callaghan, Andy Cheese, Chiyan Chen, Olaf Chitil, Javier Deniz, Tyson Dowd, Conal Elliott, Pal-Kristian Engstad, Tony Finch, Sigbjorn Finne, Richard Gomes, John Heron, Stefan Karrmann, Richard Kuhns, Ronald Legere, Phil

[11] Some smart compilers use type-based effect systems to guide their optimisers, but that is different from the programmer-visible type system.

[12] I describe them all as "language extensions" because, while none has a significant impact on Haskell's syntax or type system, all have an impact on its semantics and implementation

Molyneux, Andy Moran, Vincent Labrecque, Anders Lau Olsen, Andy Pitts, Tom Pledger, Martin Pokorny, Daniel Russell, George Russell, Tim Sauerwein, Julian Seward, Christian Sievers, Dominic Steinitz, Jeffrey Straszheim, Simon Thompson, Mark Tullsen, Richard Uhtenwoldt, Don Wakefield, and Fredrik Winkler for their extremely helpful feedback.

References

[1] ALLISON, L. *A Practical Introduction to Denotational Semantics*. Cambridge University Press, Cambridge, England, 1986.

[2] ARIOLA, Z., AND SABRY, A. Correctness of monadic state: An imperative call-by-need calculus. In *25th ACM Symposium on Principles of Programming Languages (POPL'98)* (San Diego, Jan. 1998), ACM.

[3] ARMSTRONG, J., VIRDING, R., , WIKSTROM, C., AND WILLIAMS, M. *Concurrent programming in Erlang (2nd edition)*. Prentice Hall, 1996.

[4] BARENDSEN, E., AND SMETSERS, S. Uniqueness typing for functional languages with graph rewriting semantics. *Mathematical Structures in Computer Science 6* (1996), 579–612.

[5] BENTON, N., AND KENNEDY, A. Monads, effects, and transformations. In *Higher Order Operational Techniques in Semantics: Third International Workshop* (1999), no. 26 in Electronic Notes in Theoretical Computer Science, Elsevier, pp. 1–18.

[6] BENTON, N., KENNEDY, A., AND RUSSELL, G. Compiling Standard ML to Java bytecodes. In ICFP98 [18], pp. 129–140.

[7] Bird, R., and Wadler, P. *Introduction to Functional Programming*. Prentice Hall, 1988.

[8] Chakravarty, M. C -> Haskell: yet another interfacing tool. In Koopman and Clack [23].

[9] Felleisen, M., and Hieb, R. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science 103* (1992), 235–271.

[10] Finne, S., Leijen, D., Meijer, E., and Peyton Jones, S. H/Direct: a binary foreign language interface for Haskell. In ICFP98 [18], pp. 153–162.

[11] Finne, S., Leijen, D., Meijer, E., and Peyton Jones, S. Calling Hell from Heaven and Heaven from Hell. In ICFP99 [19], pp. 114–125.

[12] Hammond, K., and Michaelson, G., Eds. *Research Directions in Parallel Functional Programming*. Springer-Verlag, 1999.

[13] Hayes, B. Finalization in the collector interface. In *International Workshop on Memory Management*, Y. Bekkers and J. Cohen, Eds., no. 637 in Lecture Notes in Computer Science. Springer Verlag, St. Malo, France, Sept. 1992, pp. 277–298.

[14] Hudak, P. *The Haskell school of expression*. Cambridge University Press, 2000.

[15] Hudak, P., and Sundaresh, R. On the expressiveness of purely-functional I/O systems. Tech. Rep. YALEU/DCS/RR-665, Department of Computer Science, Yale University, Mar. 1989.

[16] Hughes, R., and O'Donnell, J. Expressing and reasoning about non-deterministic functional programs. In *Functional Programming, Glasgow 1989*, K. Davis and R. Hughes, Eds. Workshops in Computing, Springer Verlag, 1989, pp. 308–328.

[17] Hutton, G., Ed. *Proceedings of the 2000 Haskell Workshop, Montreal* (Sept. 2000), no. NOTTCS-TR-00-1 in Technical Reports.

[18] *ACM SIGPLAN International Conference on Functional Programming (ICFP'98)* (Baltimore, Sept. 1998), ACM.

[19] *ACM SIGPLAN International Conference on Functional Programming (ICFP'99)* (Paris, Sept. 1999), ACM.

[20] Jeffery, D., Dowd, T., and Somogyi, Z. MCORBA: a CORBA binding for Mercury. In *Practical Applications of Declarative Languages* (San Antonio, Texas, 1999), Gupta, Ed., no. 1551 in Lecture Notes in Computer Science, Springer Verlag, pp. 211–227.

[21] Jouvelot, P., and Gifford, D. Algebraic reconstruction of types and effects. In *18'th ACM Symposium on Principles of Programming Languages (POPL), Orlando*. ACM, Jan. 1991.

[22] Karlsen, E. *Tool integration in a functional programming language*. PhD thesis, University of Bremen, Nov. 1998.

[23] Koopman, P., and Clack, C., Eds. *International Workshop on Implementing Functional Languages (IFL'99)* (Lochem, The Netherlands, 1999), no. 1868 in Lecture Notes in Computer Science, Springer Verlag.

[24] Launchbury, J. A natural semantics for lazy evaluation. In POPL93 [40], pp. 144–154.

[25] Launchbury, J., Lewis, J., and Cook, B. On embedding a microarchitectural design language within Haskell. In ICFP99 [19], pp. 60–69.

[26] Leijen, D., and Hook, E. M. J. Haskell as an automation controller. In *Third International School on Advanced Functional Programming (AFP'98)* (Braga, Portugal, 1999), no. 1608 in Lecture Notes in Computer Science, Springer Verlag.

[27] Marlow, S. Writing high-performance server applications in haskell. In Hutton [17].

[28] Marlow, S., Peyton Jones, S., and Elliott, C. Stretching the storage manager: weak pointers and stable names in Haskell. In Koopman and Clack [23].

[29] Marlow, S., Peyton Jones, S., and Moran, A. Asynchronous exceptions in Haskell. In *ACM Conference on Programming Languages Design and Implementation (PLDI'99)* (Snowbird, Utah, June 2001), ACM, pp. 274–285.

[30] Meijer, E., and Finne, S. Lambada: Haskell as a better Java. In Hutton [17].

[31] Milner, R. *Communicating and Mobile Systems : The Pi-Calculus*. Cambridge University Press, 1999.

[32] Moggi, E. Computational lambda calculus and monads. In *Logic in Computer Science, California*. IEEE, June 1989.

[33] Nelson, G., Ed. *Systems Programming with Modula-3*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[34] Nordin, T., Peyton Jones, S., and Reid, A. Green Card: a foreign-language interface for Haskell. In

Haskell workshop (Amsterdam, 1997), J. Launchbury, Ed.

[35] PEYTON JONES, S., GORDON, A., AND FINNE, S. Concurrent Haskell. In *23rd ACM Symposium on Principles of Programming Languages (POPL'96)* (St Petersburg Beach, Florida, Jan. 1996), ACM, pp. 295–308.

[36] PEYTON JONES, S., MEIJER, E., AND LEIJEN, D. Scripting COM components in Haskell. In *Fifth International Conference on Software Reuse* (Los Alamitos, CA, June 1998), IEEE Computer Society, pp. 224–233.

[37] PEYTON JONES, S., REID, A., HOARE, C., MARLOW, S., AND HENDERSON, F. A semantics for imprecise exceptions. In *ACM Conference on Programming Languages Design and Implementation (PLDI'99)* (Atlanta, May 1999), ACM, pp. 25–36.

[38] PEYTON JONES, S., AND WADLER, P. Imperative functional programming. In POPL93 [40], pp. 71–84.

[39] POOLE, I. Public report of the SADLI project: safety assurance in diagnostic laboratory imaging. Tech. rep., MRC Human Genetics Unit, Edinburgh, Mar. 1995.

[40] *20th ACM Symposium on Principles of Programming Languages (POPL'93)* (Jan. 1993), ACM.

[41] REPPY, J. *Concurrent programming in ML*. Cambridge University Press, 1999.

[42] ROSCOE, B. *The Theory and Practice of Concurrency*. Prentice Hall, 1998.

[43] SCHMIDT, D. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, 1986.

[44] THOMPSON, S. *Haskell: the craft of functional programming*. Addison Wesley, 1999.

[45] TOLMACH, A. Optimizing ML using a hierarchy of monadic types. In *Workshop on Types in Compilation '98, Kyoto* (Mar. 1998), Lecture Notes in Computer Science, Springer Verlag, pp. 97–113.

[46] TRINDER, P., HAMMOND, K., LOIDL, H.-W., AND PEYTON JONES, S. Algorithm + strategy = parallelism. *Journal of Functional Programming 8* (Jan. 1998), 23–60.

[47] WADLER, P. Comprehending monads. *Mathematical Structures in Computer Science 2* (1992), 461–493.

[48] WADLER, P. The essence of functional programming. In *20th ACM Symposium on Principles of Programming Languages (POPL'92)*. ACM, Albuquerque, Jan. 1992, pp. 1–14.

[49] WADLER, P. Monads for functional programming. In *Advanced Functional Programming*, J. Jeuring and E. Meijer, Eds., vol. 925 of *Lecture Notes in Computer Science*. Springer Verlag, 1995.

[50] WADLER, P. How to declare an imperative. *ACM Computing Surveys 29*, 3 (1997).

[51] WADLER, P. The marriage of effects and monads. In ICFP98 [18], pp. 63–74.

[52] WRIGHT, A., AND FELLEISEN, M. A syntactic approach to type soundness. *Information and Computation 115* (Nov. 1994), 38–94.