Abstract

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

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

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

This paper appears in the Proceedings of the 23rd ACM Symposium on Principles of Programming Languages (POPL '96), St Petersburg Beach, Florida, Jan 1996.

1 Introduction

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

This paper makes the following contributions:

- We show how concurrency can be smoothly integrated into a lazy purely-functional language, using only four new primitive operations and no new language constructs (Section 2). Perhaps surprisingly, choice is not one of these primitive operations (Section 5).
- We give numerous examples of useful abstractions that can readily be built in Concurrent Haskell (Sections 3 and 4).
- We give a semantics for Concurrent Haskell that is clearly stratified into a deterministic layer and a concurrency layer (Section 6). Existing reasoning techniques can be retained unmodified; for example, program transformations that preserve the correctness of a sequential Haskell program also preserve correctness of a Concurrent Haskell program. This is an unusual feature; more commonly, the non-determinism that arises from concurrency pervades the entire language.

Concurrent Haskell is implemented, freely available, and is the substrate upon which we are building the Haggis graphical user interface toolkit (Finne & Peyton Jones [1995]).

This paper is not at all about concurrency as a means of increasing performance by exploiting multiprocessors. Our approach to that goal uses implicit, semantically transparent, parallelism; but that is another story. Rather, this paper concerns the use of explicit, semantically visible, concurrent I/O-performing processes. Our goal is to extend Haskell's usefulness into a new class of applications.

2 The basic ideas

Concurrent Haskell adds two main new ingredients to Haskell:

- processes, and a mechanism for process initiation (Section 2.2); and
- atomically-mutable state, to support inter-process communication and cooperation (Section 2.3).

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

The semantics of Concurrent Haskell is discussed later, in Section 6.

2.1 A review of monadic I/O

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

\[
\text{type IO a = World } \rightarrow (a, \text{World})
\]

That is, a value of type \(\text{IO} a\) takes a world state as input, and delivers a modified world state together with a value of type \(a\). Of course, the implementation performs the I/O right away — thereby modifying the state of the world “in place”.

We call a value of type \(\text{IO} t\) an action. Here are two useful actions:

- \(\text{hGetChar :: Handle } \rightarrow \text{IO Char}\)
- \(\text{hPutChar :: Handle } \rightarrow \text{Char } \rightarrow \text{IO ()}\)

The action \(\text{hGetChar}\) reads a character from the specified handle (which identifies some file or other byte stream), and returns it as the result of the action: \(\text{hPutChar}\) takes a handle and a character and returns an action that writes the character to the specified file or stream.

Actions can be combined in sequence using the infix combinators \(\gg\) and \(\gg=\):

\[
\begin{align*}
\gg &: : \text{IO } a \rightarrow \text{IO } b \rightarrow \text{IO } b \\
\gg= &: : \text{IO } a \rightarrow (a \rightarrow \text{IO } b) \rightarrow \text{IO } b
\end{align*}
\]

For example, here is an action that reads a character from the standard input, and then prints it twice to the standard output:

\[
\begin{align*}
\text{hGetChar } \text{stdin} & \gg= \text{\textbackslash c} \rightarrow \\
\text{hPutChar } \text{stdout } \text{c} & \gg \\
\text{hPutChar } \text{stdout } \text{c}
\end{align*}
\]

(The notation \(\text{\textbackslash c} \rightarrow \text{E}\) for some expression \(\text{E}\), denotes a lambda abstraction. In Haskell, the scope of a lambda abstraction extends as far to the right as possible; in this example the body of the \(\text{\textbackslash c}\) abstraction includes everything after the \(\text{\textbackslash c}\).)

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

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

\[
\text{return :: a } \rightarrow \text{IO a}
\]

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

\[
\begin{align*}
\text{echo :: IO Char} \\
\text{let e = hGetChar } \text{stdin} \gg= \text{\textbackslash c} \rightarrow \\
\text{hPutChar } \text{stdout } \text{c} \gg \\
\text{return } \text{c}
\end{align*}
\]

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

\[
\begin{align*}
\text{newMutVar} &: : \text{MutVar } a \\
\text{readMutVar} &: : \text{MutVar } a \rightarrow \text{IO } a \\
\text{writeMutVar} &: : \text{MutVar } a
\end{align*}
\]

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

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

\[
\begin{align*}
\text{main :: IO } () \\
\text{main = echo } \gg= \text{\textbackslash c} \rightarrow \\
\text{if } c = \text{\"\n\"} \\
\text{then return } () \\
\text{else main}
\end{align*}
\]

In principle, then, a program is just a state transformer that is applied to the real world to give a new world. In practice, however, it is crucial that the side-effects the program specifies are performed incrementally and not all at once when the program finishes. A state-transformer semantics for I/O is therefore, alas, unsatisfactory, and becomes untenable when concurrency is introduced, a matter to which we return in Section 6.

More details of monadic I/O and state transformers can be found in Gordon [1994a], Launchbury & Peyton Jones [1994], Peyton Jones & Wandler [1993]. Other I/O mechanisms for purely-functional languages are surveyed by Gordon [1993].

### 2.2 Processes

Concurrent Haskell provides a new primitive \(\text{forkIO}\) which starts a concurrent process:\n
\[
\text{forkIO :: IO } () \rightarrow \text{IO } ()
\]

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

\[
\begin{align*}
\text{let } \\
\text{loop } ch = \text{hPutChar } \text{stdout } \text{ch} \gg \text{loop } ch \\
\text{forkIO } (\text{loop } \text{\textbackslash ch\}' = \text{echo } \gg= \text{\textbackslash c} \rightarrow \\
\text{hPutChar } \text{stdout } \text{c} \gg \\
\text{return } \text{c}
\end{align*}
\]

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

2. We use the term \textit{process} to distinguish explicit concurrency from implicit parallelism, for which we use the term \textit{thread. A process is managed by the Haskell runtime system, and certainly does not correspond to a Unix process.
loop 'z'
The forkIO spawns a process which performs the action
loop 'a'. Meanwhile, the "parent" process continues on to
perform loop 'z'. The result is that an infinite sequence of
interleaved 'a's and 'z's appears on the screen; the exact
interleaving is unspecified (but see Section 6.3).

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

```haskell
mailLoop :: IO ()
mailLoop
    = getButtonPress b >>= \ v ->
    case v of
        Compose -> forkIO doCompose >>
        mailLoop
other things
```
doCompose :: IO () -- Pop up and manage
doCompose = ... -- Composition window

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

The following features of forkIO are worth noting:

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

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

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

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

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

2.3 Synchronisation and communication

At first we believed that forkIO alone would be sufficient
to support concurrent programming in Haskell, provided that
the underlying implementation correctly handled the syn-
nchronisation between two processes that try to evaluate
the same thunk. Our belief was based on the idea that two
processes could communicate via lazy-evaluated streams,
produced by one and consumed by the other (Kahn & Mac-
Queen [1977]). Whilst processes can indeed communicate
in this way, we found at least three distinct reasons to intro-
duce additional mechanisms for synchronisation and com-
munication between processes:

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

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

3. Writing stream-processing programs is thoroughly awk-
ward, especially if a function consumes several streams
and produces several others, as well as performing in-
put/output. One of the reasons that monadic I/O has
become so popular is precisely because stream-style
I/O is so tiresome to program with. It would be ironic
if Concurrent Haskell re-introduced stream processing
for inter-process communication just as monadic I/O
abolished it for input/output! We wanted to find a way
to make communication between processes look just as
convenient as I/O; indeed, from the point of view of
any particular process the other processes might just
as well be considered part of the external world.
Our solution is to combine our work on mutable state (Lancbaury & Peyton Jones [1994]) with the L-structures and M-structures of the dataflow language Idf (Arvind, Nikhil & Pingali [1998]; Barh, Nikhil & Arvind [1991]). First of all we have a new primitive type:

```plaintext
type MVar a
```

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

```plaintext
newMVar :: IO (MVar a) creates a new MVar.
takeMVar :: MVar a -> IO a blocks until the location is non-empty, then reads and returns the value, leaving the location empty.
putMVar :: MVar a -> a -> IO () writes a value into the specified location. If there are one or more processes blocked in takeMVar on that location, one is thereby allowed to proceed. It is an error to perform putMVar on a location which already contains a value. (See Section 9 for a discussion of other possible design choices for putMVar.)
```

The type MVar can be seen in three different ways:

- It can be seen as a synchronised version of the type Mutex introduced in Section 2.1.
- It can be seen as the type of channels, with takeMVar and putMVar playing the role of receive and send.
- A value of type MVar () can be seen as a binary semaphore, with the signal and wait operations implemented by putMVar and takeMVar respectively.

MVars are also somewhat reminiscent of ML’s ref types, which require quite a bit of work in the type system to preserve soundness. It turns out that this type-soundness problem does not arise for us, because values of type MVar t can only be lambda-bound, and hence must be monomorphic.

### 3 A standard abstraction: buffering

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

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

#### 3.1 A buffer variable

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

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

```plaintext
type CVar a = (MVar a, -- Producer -> consumer MVar () ) -- Consumer -> producer
newCVar :: IO (CVar a)
```

```plaintext
newCVar = newMVar >>\ = \ data_var ->
newMVar >>\ = \ ack_var ->
p putMVar ack_var () >> return (data_var, ack_var)
```

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

```plaintext
g etCVar :: CVar a -> IO a
g etCVar (data_var, ack_var) = takeMVar data_var >>\ = \ val ->
p putMVar ack_var () >> return val
```

#### 3.2 A buffered channel

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

```plaintext
type Channel a
newChan :: IO (Channel a)
p utChan :: Channel a -> a -> IO ()
g etChan :: Channel a -> IO a
```

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

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

```plaintext
type Channel a = (MVar (Stream a), -- Read MVar (Stream a)) -- Write
```

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

```plaintext
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:

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

A Stream can therefore be thought of as a list, consisting of alternating Items and full MVars, terminated with a “hole” consisting of an empty MVar. The write end of the channel
Creating a new channel is now just a matter of creating the read and write MVars, plus one (empty) MVar for the stream itself:

\[
\text{newChan} = \text{newMVar} \quad \Rightarrow \text{\langle read \rangle} \\
\text{newMVar} \quad \Rightarrow \text{\langle write \rangle} \\
\text{newMVar} \quad \Rightarrow \text{\langle hole \rangle} \\
\text{putMVar read hole} \Rightarrow \\
\text{putMVar write hole} \Rightarrow \\
\text{return (read, write)}
\]

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:

\[
\text{putChan (read,write) val} \\
\qquad = \text{newMVar} \quad \Rightarrow \text{\langle new, hole \rangle} \\
\qquad \text{takeMVar write} \quad \Rightarrow \text{\langle old, hole \rangle} \\
\qquad \text{putMVar write new, hole} \Rightarrow \\
\qquad \text{return (new, write)}
\]

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

\[
\text{getChan (read,write)} \\
\qquad = \text{takeMVar read} \quad \Rightarrow \text{\langle cts \rangle} \\
\qquad \text{takeMVar cts} \quad \Rightarrow \text{\langle Item val new, hole \rangle} \\
\qquad \text{putMVar read new} \Rightarrow \\
\qquad \text{return val}
\]

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-est 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:

\[
\text{dupChan :: Channel a \rightarrow IO (Channel a)}
\]

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:

\[
\text{dupChan (read,write)} \\
\qquad = \text{newMVar} \quad \Rightarrow \text{\langle new, hole \rangle} \\
\qquad \text{takeMVar write} \quad \Rightarrow \text{\langle old, hole \rangle} \\
\qquad \text{putMVar write new, hole} \Rightarrow \\
\qquad \text{return (new, write)}
\]

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

\[
\text{unGetChan :: Channel a \rightarrow a \rightarrow IO ()}
\]

### 3.3 Skip channels

As a final example, Figure 2 implements a *skip channel*, a useful abstraction that we have not seen elsewhere in the literature. A skip channel is useful when an intermittent source of high-bandwidth information (mouse-movement events, for example) is to be coupled to a process that may only be able to deal with events at a lower rate (scrolling a window, for example). A read operation on a skip channel either returns the most-recently-written value (skipping any values written previously), or else blocks if no write has been performed since the last read. To make it more interesting, a dupSkipChan operation is also provided that allows multiple independent readers, each with the above semantics.

A skip channel is implemented as a pair of MVars. The second is a semaphore; it is full if the skip channel contains a value as yet unread by this reader, and empty otherwise. The first contains a pair consisting of the current contents of the channel and a list of the empty semaphores of the readers that have already read the channel's current contents. With this in mind the implementation of the skip channel's operations should be easy to follow.
4 Control over scheduling

Next we study some examples that demonstrate how it is possible to "reify" scheduling decisions, allowing the programmer to take control of them. Suppose we wanted to implement a channel with bounded buffering; that is, one in which the writer would block if there were more than a certain number of unread elements in the buffer. A straightforward way to implement a bounded channel would be as a pair of an unbounded channel and a quantity semaphore:

\[ \text{type BChannel } a = (\text{Channel } a, \text{QSem} ) \]

A quantity semaphore is an abstraction with the following interface:

\[ \begin{align*}
\text{type QSem} & \quad :: \text{IO} \ QSem \\
\text{newQSem} & \quad :: \ QSem \to \text{IO} (\text{QSem}) \\
\text{waitQSem} & \quad :: \ QSem \to \text{IO} () \\
\text{signalQSem} & \quad :: \ QSem \to \text{IO} () \\
\end{align*} \]

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

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

4.1 Implementing quantity semaphores

It is possible to implement a quantity semaphore using only binary semaphores, but it is surprisingly difficult, and correct solutions are not well known (Barz [1983]). However, because we can freely allocate new \( \text{MVar}s \), we can give a perfectly straightforward implementation:

\[ \begin{align*}
\text{type QSem} & \quad :: \text{MVar} (\text{Int}, [\text{MVar} ()]. \text{MVar} () ) \\
\end{align*} \]

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

If a waitQSem finds a zero count in the QSem, it creates a new, private, \( \text{MVar} \), adds it to the list, puts the resulting pair back in the QSem’s \( \text{MVar} \), and then blocks on its private \( \text{MVar} 

\[ \begin{align*}
\text{waitQSem sem} & \quad :: \text{QSem} \to \text{IO} () \\
\text{signalQSem sem} & \quad :: \text{QSem} \to \text{IO} () \\
\end{align*} \]

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

\[ \begin{align*}
\text{signalQSem sem} & \quad :: \text{QSem} \to \text{IO} () \\
\text{case blkd of} & \quad :: \text{IO} () \\
[] & \quad \to \text{putMVar} (\text{avail}+1, []) \\
(\text{blk}:\text{blkd'}) & \quad \to \text{putMVar} (0, \text{blk}:\text{blkd'}) \\
\text{putMVar blk} & \quad :: \text{IO} () \\
\end{align*} \]

The implementation of \text{waitQSem} might be implemented in this way, deadlock might easily result. Why? Because two processes executing a waitQSem might each claim part, but not all, of the resource they require, thereby depleting it to zero and deadlocking. So waitQSem must grab all its requirement at once; if not enough is available, it must block without grabbing any.

4.2 Variable-munch quantity semaphores

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

\[ \begin{align*}
\text{waitQSem n} & \quad :: \text{QSem} \to \text{Int} \to \text{IO} () \\
\text{signalQSem n} & \quad :: \text{QSem} \to \text{Int} \to \text{IO} () \\
\end{align*} \]

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

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

\[ \begin{align*}
\text{type QSem} & \quad :: \text{MVar} (\text{Int}, [\text{MVar} ()]) \\
\end{align*} \]

The implementation of \text{waitQSem n} is essentially identical to \text{waitQSem}, \text{signalQSem n} is a bit more interesting, because it
may free zero or more blocked processes:

```haskell
signalQSemN sem n
= takeMVar sem >>=
  \(\text{avail}, \text{blkd}\) \rightarrow
  \(\text{avail}', \text{blkd}'\) \rightarrow
  \(\text{putMVar sem (avail'), (blkd')}\)
free avail [\(\text{req}, \text{blk}\): \(\text{blkd}\)]
  = if avail \geq req then
    \(\text{putMVar blk (}) \rightarrow
      \text{free (avail'-req), (blk')}\)
  else
    \\
  \(\text{free avail blk} \geq \(\text{avail}', \text{blkd}'\) \rightarrow
    \text{return (avail', (req, blk'), (blkd'))}\)
```

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

### 4.3 Priority

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

### 4.4 Summary

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

### 5 Choice

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

\[
x(v).P + y(w).Q
\]

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

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

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

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

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

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

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

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

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

### 5.1 Iterated choice

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

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

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

```
ge1, ge2 :: IO Packet
get1, ge2 get :: Packet -> IO ()
```
Of course, get1 and get2 can be as complicated as necessary. They might consist of a large series of I/O interactions, not just one primitive operation.

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

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

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

    server_loop buf

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

suck :: IO a -> CVar a -> IO ()
suck get_op buf
  = get_op
    >>= \pkt ->
    putCVar buf pkt >>
    suck get buf
```

Of course, if the clients can be "told" how to write to the server the "sucking" processes are not necessary. In practice we find that this approach, which is strongly reminiscent of call-backs, loses a degree of modularity — for example, the client would have to be informed if the server changes — so we normally use the formulation given above.

5.2 Singular choice

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

```haskell
type Alternative a = Commitment a -> IO ()

data Maybe a = Nothing | Just a
```

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

It is now simple to define select:

```haskell
select :: [Alternative a] -> IO a
select arms
  = newMVar >>=
    \commit_var ->
    newMVar
    >>= \result_var ->
    putMVar commit_var
    Nothing >>
    return res
    let
doe_arm arm = forkIO (arm commit)
in
    mapIO doe_arms
    takeMVar result_var
```

Here, mapIO is an analogue in the IO monad of the familiar map function:

```haskell
mapIO :: (a -> IO b) -> [a] -> IO [b]
mapIO f xs produces an IO action in each case. It performs these actions in sequence, and returns the list of their results.
```

6 Semantics

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

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

Instead we adopt an operational semantics, the standard approach to giving the semantics of a concurrent language.

6.1 Deterministic Reduction

Suppose we already have an operational semantics for a purely functional fragment of Haskell. Gordon [1994b] presents a suitable operational semantics for a small fragment of Haskell, and the approach could be extended to the full language.

We shall show how to incorporate our concurrency primitives into such a semantics. Suppose \(A\) and \(B\) stand for types and \(a\) and \(b\) stand for programs, that is, closed, well-typed expressions, and that the operational semantics consists of a deterministic, small-step reduction relation, \(a \rightarrow b\).

We extend the grammar of types by

\[
A ::= \ldots | \text{MVar} A | IO A
\]

3 The Maybe type is standard in Haskell, and corresponds to option in Standard ML. A value of type Maybe \(t\) is either Nothing or is of the form Just \(v\), where \(v\) has type \(t\). Maybe types are useful for encoding values which may or may not be there.
and allow the following new constants as expressions:

\[
\begin{align*}
\text{return} & \quad >>\\text{/}\ \\
forkIO & \quad \text{newMVar} \\
\text{putMVar} & \quad \text{takeMVar}
\end{align*}
\]

A name, \( n \), is drawn from an infinite set of tags, and uniquely identifies a particular \( \text{MVar} \). We extend the reduction relation to reduce the first argument of \((\gg\gg)\) and of \( \text{putMVar} \) and \( \text{takeMVar} \), and with the following axiom scheme

\[
\text{return } a \gg\gg b \iff b(a)
\]

but we do not provide any reductions for \( \text{forkIO} \), \( \text{newMVar} \), \( \text{putMVar} \) and \( \text{takeMVar} \). It follows that a \text{value} — that is, a fully reduced program of type \( \text{IO } A \) — is either \text{return } a where \( a : A \) or of the form \( \mathcal{M}[v_{\text{IO}}] \) where

\[
v_{\text{IO}} \ ::= \forkIO \cdot \text{newMVar} | \text{putMVar} \cdot n | \text{takeMVar} \\
\mathcal{M}[n] \ ::= [] | \mathcal{M}[] \gg\gg a
\]

In a value \( \mathcal{M}[v_{\text{IO}}] \), the expression \( v_{\text{IO}} \) represents the next concurrent action, and the context \( \mathcal{M}[n] \) represents the continuation that consumes the result of that action. This mild extension preserves determinacy of \( \equiv \).

### 6.2 Concurrent Reaction

To model the concurrent aspects of Concurrent Haskell we need to consider systems of interacting monadic processes. We use \( P \) and \( Q \) to stand for processes.

\[
P \ ::= \quad a \quad \text{if} \ a : \text{IO}(i)
\]

\[
\begin{align*}
P | Q & \quad \text{parallel composition} \\
(\text{in}) \cdot P & \quad \text{restriction of name } n \text{ to } P \\
(\alpha) \cdot n & \quad \text{full } \text{MVar} \text{ named } n \text{ holding program } a \\
\hat{\alpha} & \quad \text{empty } \text{MVar} \text{ named } n \\
\text{ABORT} & \quad \text{erroneous process}
\end{align*}
\]

The only binding construct for names is \( (\text{in}) \cdot P \). We write \( \text{fn}(P) \) for the set of names free in process \( P \), and \( P[\text{in}/n] \) for the outcome of substituting \( n \) for each occurrence of name \( n \) free in process \( P \).

We adopt the 'chemical abstract machine' presentation of polyadic π-calculus [Milner 1991]. First, we formalise the idea of a 'solution' of programs and \( \text{MVars} \) waiting to be acted on by defining a structural congruence relation. Second, we specify the reaction of programs and \( \text{MVars} \) by simple reaction rules.

Let \text{structural congruence}, \( \equiv \), be the least congruence (that is, an equivalence relation preserved by all process contexts) to include alpha-conversion of bound variables and names, plus the following two collections of rules. The first group says that a process solution is roughly a multiset:

\[
\begin{align*}
P_1 | (P_2 | P_3) & \equiv (P_1 | P_2) | P_3 \\
P | Q & \equiv Q | P
\end{align*}
\]

The second group are the standard rules for restriction from \( \pi \)-calculus. Restriction represents the locality of access of \( \text{MVars} \).

\[
\begin{align*}
(\text{in}) \cdot (\text{in}) \cdot P & \equiv \text{in} \cdot (\text{in}) \cdot P \\
(\text{in}) \cdot P | Q & \equiv P | \text{in} \cdot Q, \text{ if } n \notin \text{fn}(P)
\end{align*}
\]

Secondly, we extend the deterministic reduction relation \( \rightarrow \), on programs to a nondeterministic reaction relation, \( \rightarrow \), on processes, identified up to structural congruence. The first two rules specify the interaction of programs and \( \text{MVars} \):

\[
\begin{align*}
(\text{Put}) & \quad \alpha \cdot n | \mathcal{M}[\text{putMVar} \cdot a] \rightarrow (\alpha) \cdot n | \mathcal{M}[(\text{return})] \\
(\text{Take}) & \quad (\alpha) \cdot n | \mathcal{M}[\text{takeMVar}] \rightarrow \alpha \cdot \mathcal{M}[(\text{return})] \\
(\text{Abort}) & \quad (\alpha) \cdot n | \mathcal{M}[\text{return} \cdot \text{abot}] \rightarrow \text{ABORT}
\end{align*}
\]

The (Abort) rule deals with the erroneous situation of a \( \text{putMVar} \) on a full \( \text{MVar} \). We also need two rules to deal with the propagation of \( \text{ABORT} \):

\[
\begin{align*}
(\text{AbortPar}) & \quad \text{ABORT} | P \rightarrow \text{ABORT} \\
(\text{AbortNu}) & \quad (\text{in}) \text{ABORT} \rightarrow \text{ABORT}
\end{align*}
\]

The operations \( \text{forkIO} \) and \( \text{newMVar} \) turn into process restriction and composition:

\[
\begin{align*}
(\text{Fork}) & \quad \mathcal{M}[\text{forkIO} \cdot a] \rightarrow a | \mathcal{M}[\text{return} \cdot \text{io}] \\
(\text{New}) & \quad \mathcal{M}[\text{newMVar}] \rightarrow (\text{in}) | (\alpha) \cdot n | \mathcal{M}[\text{return} \cdot \text{io}] \\
& \quad \text{if } n \notin \text{fn}(\mathcal{M})
\end{align*}
\]

These two structural rules allow reactions within compositions and beneath restrictions:

\[
\begin{align*}
(\text{Par}) & \quad P | Q \rightarrow P' | Q \quad \text{if } P \rightarrow P' \\
(\text{Res}) & \quad (\text{in})P \rightarrow (\text{in})P' \quad \text{if } P \rightarrow P'
\end{align*}
\]

The final reaction rule turns a reduction of a program into a reaction of that program considered as a process:

\[
\begin{align*}
(\text{Reduce}) & \quad a \rightarrow b \quad \text{if } a \equiv b
\end{align*}
\]

Since processes are identified up to \( \equiv \), we may freely use the rules of \( \equiv \) to bring together partner programs and \( \text{MVars} \) for (Put) or (Take) interactions, and to enlarge the scope of an \( \text{MVar} \) allocated by (New).

Our semantics is intentionally minimal but nonetheless it does support at least the following result. Say that a process \( P \) passes a test \( R \) if \( 3(Q | P | R \rightarrow^* \text{done}(Q)), \) where \( \text{done} \) is a new process constant allowed only in test processes such as \( R \). Then two processes are \text{testing equivalent} if they pass the same tests. This is a standard definition from concurrency theory (de Nicola & Hennessy [1983]).

**Theorem.** If two programs \( a \) and \( b \) are \text{denotationally equivalent} as functional programs, they are \text{testing equivalent} when considered as processes.

Our denotational semantics is a standard denotational semantics for a lazy functional language, with the \( \text{IO} \) type modelled as if it were an algebraic type with a constructor corresponding to each of the constant \( \text{putMVar} \), \( \text{takeMVar} \), \( \text{forkIO} \), \( \text{newMVar} \) and \( \text{return} \). These constants and \( \gg\gg \) are modelled by functions acting on this algebraic type. To model the values held by \( \text{MVar} \)'s we use dynamic types. We omit the details but this is a generalisation of constructions from Crole & Gordon [1994] and Gordon [1994a]. In effect, we model a program of \( \text{IO} \) type as a potentially infinite tree, where each node represents an instruction to be interpreted at runtime. The nodes representing \( \text{forkIO}'s \) have two successors, to be interpreted in parallel; all the others have one or none. We omit the proof of the theorem, but intuitively it holds because as far as passing a test is concerned, all that matters about a program of \( \text{IO} \) type is the sequence of instructions it issues. If two programs are denotationally
equivalent, they issue the same sequence of instructions, so they are testing equivalent.

This is not a particularly abstract denotational semantics, since it explicitly represents the instructions issued by a program, rather than their observable effect. However, it shares with standard denotational semantics of lazy functional languages the property that a program of any type either equals a value of that type, or denotes ⊥. This fact makes it straightforward to validate conventional reasoning about functional programs, such as βη-equivalence. In particular, the theorem asserts that any compiler optimisation that depends on such conventional reasoning will not invalidate testing equivalence.

The Concurrent Haskell type system restricts the possibility of side-effects, so we have been able to put all the work of explaining side-effects into explaining IO types. A denotational semantics for a language with unrestricted side-effects — see Crole & Gordon [1994], for instance — would need to account for side-effects at every type, and hence in general βη-equivalence (for example) is unsound.

6.3 Fairness

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

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

6.4 Summary

There have been several previous semantics for concurrent functional languages (Holmström [1983]; Jeffrey [1995]; Reppy [1992]; Schölz [1995]). Schölz’s set-based semantics is closest, but nothing in his semantics corresponds to our restriction, (pm), which captures locality of MVars.

A notable feature of our semantics is its stratification into a deterministic reduction relation □, and a non-deterministic reaction relation →. We might consider → as specifying an imperative coordination language, and □ as specifying a functional computation language.

Our semantics is sufficient to show that the non-deterministic, concurrent computation (→) at IO types does not affect the deterministic, functional computation (□) at non-IO types. We sought the simplest semantics that would do so. We have not gone further — for instance, by seeking to approximate testing equivalence using a labelled transition system and bisimilarity — because the presence of both higher-order functions and local names is known to make bisimilarity problematic. Jeffrey [1995] studies weak bisimilarity for a monadic concurrent language similar in spirit to Concurrent Haskell but does not consider the problems of local names. Although an adaptation of Jeffrey’s work to Concurrent Haskell would be a worthwhile research project, our minimal semantics suffices for many practical purposes. It provides a simple, precise and abstract specification of the operational behaviour of Concurrent Haskell programs.

7 Implementation

We have implemented Concurrent Haskell as a small extension to the Glasgow Haskell Compiler (GHC), a highly-optimising compiler for Haskell.

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

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

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

7.1 Other primitives

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

\[
\text{waitInputFD} :: \text{Int} \rightarrow \text{IO} ()
\]

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

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

\[
\text{delay} :: \text{Int} \rightarrow \text{IO} ()
\]
7.2 Garbage collection

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

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

1. A runnable process cannot be garbage collected, because it might perform more I/O.

2. A process blocked on an MVar cannot be garbage-collected if that MVar is not accessible from another non-garbage process. Why? Because the blocked process can only be released if another process puts a value into the blocking MVar, and that certainly can’t happen if the MVar is unreachable from any non-garbage process.

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

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

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

7.3 Distributed implementation

We are working on a distributed implementation of Concurrent Haskell. One nice property of MVars is that they seem relatively easy to implement in a distributed setting, compared to generalised choice for example.

Each MVar resides in one place, and a putMVar or getMVar operation on a remote MVar is implemented with a message send. The message for a getMVar carries with it the identity of the sending process, and may be blocked indefinitely at the far end, on an empty MVar. When the MVar is written to, the blocked getMVar message is returned to the sender, now carrying the value written to the MVar. On arrival at the original sender, the reply awakens the process whose identity it carries.

A putMVar message is simpler, since it requires no reply. Either it succeeds in writing to an empty MVar, or it finds a full MVar, which is a run-time error (but see Section 9).

8 Related work

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

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

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

8.1 Concurrent functional languages

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

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

type ‘a chan

type ‘a event

val receive : ‘a chan -> ‘a event

val transmit : ‘a chan -> ‘a -> unit event

val guard : (unit -> ‘a event) -> ‘a event

val wrap : (‘a event * (‘a -> ‘b)) -> ‘b event

val choose : ‘a event list -> ‘a event

val sync : ‘a event -> ‘a

receive and transmit are the primitive events, guard and wrap add pre-synchronisation and post-synchronisation ac-
tions respectively to an event, choose combines a list of events into a single event, and sync actually synchronises on an event. In many ways, a CML value of type event t is rather like a Haskell I/O action of type IO t; both are first-class values that can be synchronised on (resp. performed) repeatedly.

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

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

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

Using Gofer, Jones & Hudak [1993] have recently explored issues similar to Concurrent Haskell, introducing a (symmetric) fork primitive and synchronous channels into a monadic setting. This work differs from ours in that the emphasis is on expressing parallel algorithms succinctly rather than writing concurrent programs that engage in messy interaction with the outside world. Evaluating two monadic subcomputations in parallel, by "sparking" them using a symmetric fork primitive is convenient for many parallel algorithms, but this synchronous view of process is not appropriate in the concurrent case (see Section 2.2). Communication between these 'sparked' processes is done on exclusive, synchronous channels, considering it an error when more than one send occurs on a channel without a matching receive. This restriction is quite severe in a concurrent setting, as resource managers such as a window system that encapsulate and provide controlled access to some shared resource, cannot be readily expressed.

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

8.2 Functional operating systems

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

Cupitt’s made an advance over stream processing by introducing a form of monadic I/O (actually presented using continuations), with explicit process forking much like fork10 (Cupitt [1992]). Communication between processes was solely by sending messages to the process; that is, every process had but a single input port through which it had to multiplex all its communication.

8.3 Concurrent object-oriented languages

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

8.4 Synchronous vs asynchronous

We are convinced that an asynchronous model of communication gives a simpler, cleaner design than a synchronous one. Briefly, our reasons are as follows:

- The asynchronous model allows one to think either in terms of messages or in terms of shared memory. The synchronous model makes the former much easier than the latter, by requiring a shared memory location to be modelled by a process and associated communication protocol.
- The asynchronous model seems to be much less prone to deadlock with process creation, by substituting “passive” MVars for active processes.
- A synchronous model absolutely requires choice, with the difficulties discussed earlier, while the asynchronous model does not.
- In a distributed system, the underlying infrastructure directly supports asynchronous messages, while synchronous ones have to be programmed on top. In this sense, asynchronous communication is more primitive.

9 Conclusions and further work

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

The current semantics of MVars specify that a putMVar that finds a full MVar is an error that aborts the whole program. Several other design choices are also reasonable:
• Make an `MVar` hold a multiset of values, as in Pict channels.
• Make an `MVar` hold a sequence of values.
• Make an `MVar` hold a single value, but specify that a putMVar on a full `MVar` should block, rather than cause an error.

We are undecided whether any of these choices are “better” than our current semantics. The semantics of each is fairly easy to describe, and their implementations are not hard either.

One obvious topic for further work is further development of the formal semantics of Concurrent Haskell. On the implementation side we are actively working on a distributed, multiprocessor implementation.

Concurrent Haskell is freely available by FTP. (Connect to `ftp.dcs.glasgow.ac.uk` look in pub/haskel11/glasgow, and grab any version of Glasgow Haskell from 0.24 or later.)

Acknowledgements

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

References

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


HW Barz [Feb 1983], “implementing semaphores by binary semaphores,” *SIGPLAN Notices* 18, 30–45.


Simon B Jones [Sept 1984], “A range of operating systems written in a purely functional style,” TR 16, Department of Computer Science, University of Stirling.


J. H. Reppy [1995], “First-class synchronous operations,” in Theory and Practice of Parallel Programming (TPPP), Sendai, Japan, Springer Verlag LNCS.

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

E. Scholz [June 1995], “Four concurrency primitives for Haskell,” in The Haskell Workshop, La Jolla, P. Hudak, ed., 1–12.
