Concurrent Programming in HASKELL*

Víctor M. Gulías, José L. Freire
{gulias,freire}@dc.fi.udc.es

LFCIA, Department of Computer Science
University of La Coruña, SPAIN

Abstract

In this paper, an extension of the non-strict functional language HASKELL [3] is presented. It performs the evaluation of expressions by using multiple threads. The synchronous model and the event interface used are similar to Reppy's CML [11], but in this case we must deal with new factors like non-strictness, embedding the synchronous operations into the `IO` monad. Lazy structures are useful to model the behaviour of processes, in the same way as Hoare's traces in CSP [5]. A great attention is paid to higher-order concurrent abstractions which simplify programmer's work. A prototype implementation has been developed in YALE HASKELL.

Keywords: Functional Programming, Concurrency, Higher-Order Functions, Synchronous Communication.

1 Introduction

Conventional languages are oriented towards an execution model typically sequential, motivated by the Von Neumann architecture. In this model, a program is viewed as a stream of instructions which must be executed one after another affecting a global state. The solution of a problem using a sequential algorithm might be, in many cases, not appropriated. The real world is composed of many entities executing "their programs" at the same time. At some point, these programs communicate one another. To mimic this behaviour, the programmer has to appeal to artificial structures that complicate the algorithm. If the compiler is able to offer some concurrency support to the programmer, the modelization of the word might be done simpler.

Multiple execution points in an imperative language make the programs hard to understand and debug, and completely impossible to verify due to side-effects. Contrary to imperative languages, functional languages, particularly those with non-strict semantics and an absence of side-effects, have often been suggested as powerful tools for programming parallel computer systems. Different subexpressions can be evaluated in parallel

*Supported by *Xunta de Galicia* `XUGA10502B94`

given that the semantics of the language guarantees that no interference among them can arise. This is the sort of parallelism that Jones and Hudak [9] call *implicit parallelism*. Algorithms are written in a declarative way without taking care of the execution model. Thus, the expressions to evaluate have to be scheduled by the compiler in both time (when can a subexpression be evaluated?) and space (which processor should evaluate the expression?).

Sometimes it is difficult to exploit the implicit parallelism in a suitable way [7]. On the other hand, there is another kind of parallelism called *explicit parallelism*, in which the programmer *explicitly* requests the runtime system the concurrent evaluation of an expression, or communication/synchronization among threads. This user-driven parallelism, which we are going to call *concurrent programming*, will be the focus of this work. The extension proposed has been strongly influenced by some of the ideas used in the programming language OCCAM [8] and Reppy's concurrent ML [11], which use Hoare's CSP [5] as theoretical background.

2 Spawning `IO` Actions

In a single-threaded HASKELL program, the computation can be viewed as the demand of an `IO` action which forces the evaluation of some data. The natural extension to multiple lightweighted processes is to consider each thread as a different `IO` demand. The programmer has explicit control over the threads as well as the interaction among them. The whole computation is performed by threads which together constitute the concurrent application.

2.1 Spawning a Thread

The `spawn :: IO a -> IO (Thread a)` function creates a new `IO` demand which evaluates the `IO` action concurrently to the caller's demand (asymmetric fork). The abstract datatype `Thread a` encapsulates all the information related to a different `IO` demand. Some of the information associated with the thread object is the result of the evaluation (a MultiLisp-like future, see section 2.3.2), and an event that is triggered when the thread finishes its computation (see section 3).

The following program forks a thread to print a message concurrently with the main process:

```
main =
  spawn (print "Child process") >>
  print "Main thread"
```

All the threads are running (logically at least) in parallel. Each thread may be: (a) *Active*, the thread is ready to gain the CPU and continue its execution. The executions of all active threads are interleaved in a non-deterministic way; (b) *Waiting*, the thread is blocked trying to engage in an event, to seize a resource, or to perform an I/O operation. As soon as the thread accomplishes the action that forces him to wait, it will be returned to the active state to continue its execution; and (c) *Finished*, A thread that has finished its `IO` demand.

Since the parent and the child threads may mutate the same shared world, non-determinism is introduced. Although a part of the world can be duplicated to avoid dangling side-effects between the threads (for instance, changing operating system environment variables), this problem is not avoidable in general, and is usually desirable in interactive systems. So, mechanisms to control the access to the shared state are needed and a proof-obligation by the programmer required.

A thread is automatically terminated as soon as its `IO` action is computed, and then the result is made available to the rest of the program. With some restrictions, threads can be terminated if they are no longer useful to the rest of the program (for instance, if it is blocked in a `never` event, or blocked in a channel that is no longer used by any other thread). In the event of a runtime error in any thread, the whole concurrent program is aborted. `IO` exceptions can be catched using `handle`: `spawn (`*op* `` `handle` `` *handle*`)`.

2.2 Result of a Thread

A thread computing an `(IO a)` action is denoted by the type `(Thread a)`. A thread can access to the result produced by another thread using `getResult :: Thread a -> a`. If a thread needs the result (i.e, tries to force it), it has to wait until the value is available, so the thread result behaves in a similar way to usual lazy values[1].

In the following example, the contents of several files are read concurrently. Given the name of a file, `read` returns its contents as a string. Each thread executes `read` with a different filename. The parent thread folds all the results with the concatenation operator `++`.

```
read :: String -> IO String
read file =
   openFile ReadMode file >>= \hdl  ->
   hGetContents hdl       >>= \txt ->
   closeFile hdl          >>
   return txt
```

```
readFiles :: [String] -> IO String
readFiles fs =
  sequence [ spawn (read f) | f <- fs ]
                             >>= \threads ->
  return (foldl (++) []
            [ getResult t | t <- threads ])
```

In order to synchronize on the termination of a thread, `wait :: Thread a -> IO ()` can be used. A symmetric fork can be build as follows:

[1] In fact, our prototype takes advantage of the actual manipulation of lazy values to implement synchronization points among threads

```
symFork :: IO a -> IO b -> IO (a,b)
symFork a b =
   spawn a       >>= \threadA ->
   b             >>= \resB    ->
   wait threadA  >>
   return ( getResult threadA, resB )
```

2.3 Thread Interaction

Besides the values shared due to the lexical scope (which are non-mutable values), some means have to be provided to make possible a richer interaction among different threads: *Syncronous Channels* and *Write-Once Variables*.

2.3.1 Synchronous Channels

Communication among threads is performed using synchronous channels, a point-to-point communication in CSP-style. Channels carrying values of type `a` are denoted by `Channel a`.

When a thread wants to communicate on a channel, it must rendezvous with another thread that wants to do a complementary communication on the same channel, as other CSP-style languages. The data is not buffered although there is a queue of different threads blocked, trying to use the channel. Notice that all of the values transmitted on a single channel are required to have the same type. In contrast to most of non-functional concurrent languages as OCCAM, we allow arbitrary types to be passed down a channel, including lists, functions, arbitrary data structures, `IO` actions, and even other channels.

Using `channel :: Channel a`, a channel can be created. `accept :: Channel a -> IO a` and `send :: Channel a -> a -> IO ()` are used to deal with channel input and output, respectively.

2.3.2 Write-Once Variables

In addition to channels, write-once variables are also supplied. A write-once variable is a special cell which can be written only once. Any attempt to read the value before it is available will block the thread. These cells are similar to Reppy's condition variables or I-structures in ID. They might be used to implement MultiLisp-like futures [4]. Multiple writes into a write-once variable are not allowed.

A `writeOnce` variable is a three-tuple, composed by the value stored in the cell, an event (see section 3) which is triggered when the value is available and an action to set the value.

```
type WriteOnce a = (a, Event a, a -> IO ())

writeOnce :: IO (WriteOnce a)
```

The result of a thread is implemented using write-once variables as shown in the following example. The primitive `spawnPrim :: IO a -> IO ()` creates a new demand for the given `IO` action:

```
spawn op =
    writeOnce >>= \( a, _, set ) ->
    spawnPrim ( op >>= set ) >>
    return (Thread a)
```

3 Event Based Interaction

The previous section presents different mechanisms which allow the threads to interact one another. Sometimes, a more complex interaction is required. In CML, the notion of *first-class synchronous operations*, an implementation of Hoare's events, was introduced.

An *event* is a description of a synchronous operation, such as an operation on a channel; it is represented as a value of type `Event` a. Some basic events are provided, as well as event combinators to define more complex relations.

Note that each event name denotes an event *class*; there may be many occurrences of events in a single class, separated in time. The difference between an event and the occurrence of the event is similar to the difference between an `IO` action and the actual execution of that action.

3.1 Basic Events

- *Trivial Events.* The event that never happens `never :: Event a`, and the event that always happens `always :: a -> Event a`.
- *Channels.* Two events: (a) `receive :: Channel a -> Event a`, to get a value from a channel, and (b) `transmit :: Channel a -> a -> Event ()`, to output a value. `accept` and `send` are the synchronous versions of `receive` and `transmit`, respectively.
- *Timers.* `timeout :: Int -> Event ()` is an event that is triggered n units of time after synchronization. `timer :: Int -> IO (Event ())` is an action that delivers an event that is triggered n units of time after executing it.
- *Other Events.* An event handler is provided with write-once variables and threads (using `isFinished :: Thread a -> Event ()`). These events are triggered when their values are available. That can be useful to block a thread until the required values are ready to be used. Other basic events can be implemented such as user interface events, network events and many other stimuli which are interesting for the reactive system.

3.2 Event combinators

Simpler events can be combined to form more complex ones. Two different combinators are provided: alternative selection and wrappers.

- *Alternative Selection of Events.* Sometimes it is interesting to define an event as the non-deterministic selection of many possible asynchronous events. Typical CSP-like languages, such as AMBER [1] or OCCAM, include an alternative selection from multiple sources. Two combinators are included: `choose, choosePrio :: [Event a] -> Event a`. The `choose` event combines a list of events in a unique event. The result is the selection of the first event that happens, or the non-deterministic selection whenever multiple events occur simultaneously. The non-deterministic selection is not appropriated in many cases, hence a priority scheme should be introduced. `choosePrio` uses the order in the list to select an event in the case that multiple events happen.

- *Wrapping.* The `(==>) :: Event a -> (a -> b) -> Event b` combinator applies a function to the result of an event. It is specially useful to unify the type of many events to build a `choose` event.

3.3 Engaging on Events

An `Event` value describes a potential synchronous operation. Two functions are provided to let the thread synchronize on an event: (a) `poll :: Event a -> IO (Maybe a)` first establishes whether the event can or cannot be commited. If the event can be commited inmediately, `poll` carries out the synchronization, returning the value wrapped with the constructor `Just`[2]. Otherwise, the `Nothing` constructor is inmediately returned; and (b) `sync :: Event a -> IO a`, which is the blocked version of `poll`. If the event cannot be commited inmediatly, the thread will be forced to wait until the event happens.

3.4 Event Abstractions

3.4.1 Operations on Channels

The synchronous operations on channels can be implemented using the events `receive` and `transmit`. In the example, the infix operator `(.)` denotes the composition of functions:

```
accept = sync . receive
send c = sync . transmit c
```

[2]`Maybe` is a datatype defined in HASKELL's prelude as `data Maybe a = Nothing | Just a`

3.4.2 Selection of multiple events

As said, CSP-like languages usually include a *select* sentence for alternative selection from multiple choices. That sentence can be implemented as follows:

```
sel :: [Event a] -> IO a
sel = sync . choose
```

The following code shows a server with two channels: `c1` is used to output its current state, `c2` is used to receive a function to update its state. The non-deterministic selection is used to determine the next state.

```
server c1 c2 s =
  sel [ transmit c1 st ==> const s,
        receive c2     ==> \f -> f s
      ] >>= \s' ->
  server c1 c2 s'
```

3.4.3 Synchronization on a set of events

It is often needed to synchronize on all the events from a given set. Sometimes, the relative order of the events is known, but sometimes the events can happen in non-deterministic order. `syncSequence` and `syncAccumulate` engage in all the events in a given list in sequence, while `syncAll` makes it so in non-deterministic order:

```
syncSequence :: [Event a] -> IO ()
syncSequence = sequence . (map sync)

syncAccumulate :: [Event a] -> IO [a]
syncAccumulate = accumulate . (map sync)

syncAll :: [Event a] -> IO ()
syncAll es =
  accumulate [ spawn (sync e)
                   | e <- es ] >>= \ts ->
  syncSequence [ isFinished t | t <- ts ]
```

In Hoare's work, traces are used to describe the behaviour of a process by signaling the sequence of events which the process engages on. `trace` creates a new thread which will engage on a sequence of events. Thread execution is finished as soon as the thread reaches the end of the list.

```
trace :: [Event a] -> IO (Thread ())
trace = spawn . syncSequence
```

Infinite process, i.e., threads that engages on an infinite number of events, can be modelled using lazy lists as traces. All the prelude functions on lists are available to perform trace operations. For example, `repeat :: a -> [a]` creates an infinite trace by repeating the same event; `cycle :: [a] -> [a]` creates an infinite trace by cyclicing a list of events; or `(++) :: [a] -> [a] -> [a]` creates a new trace as the sequence of two traces.

Another useful abstraction is to think in the result of the event as a finalization condition. `syncWhile` takes a trace of `Event Bool` and engages on all the events while all the results are `True` and there are more events.

```
sequenceWhile :: [IO Bool] -> IO ()
sequenceWhile ops = loop ops True
 where loop _        False = return ()
       loop []       _     = return ()
       loop (op:ops) True  = op >>= loop ops

syncWhile :: [Event Bool] -> IO ()
syncWhile = sequenceWhile . (map sync)
```

For example, the following function reads values from the channel c until the value x is received:

```
t :: (Eq a) => a -> Channel a -> IO ()
t x c = syncWhile
           (repeat (receive c ==> (/= x)))
```

4 Dinning Philosophers

This is a classical problem of communicating sequential processes which is going to be implemented using this extension. A detailed description of this problem and its solution in CSP is presented in [5]. A simpler implementation might be done considering each fork as a one bit buffer. Given a fork, `pickUp` and `putDown` provides the philosopher's events related to this fork. The one bit buffer is a process that cyclically transmits and receives a value from the channel. In this case, the value is *per se* irrelevant (we are only interested in the synchronization), so $\perp$ is used.

```
type Fork = Channel ()

fork :: IO Fork
fork =
   channel >>= \c ->
   trace (cycle [ transmit c _,
                  receive c ])  >>
   return c

pickUp :: Fork -> Event ()
pickUp = receive
```

```
putDown :: Fork -> Event ()
putDown f = transmit f _
```

Each philosopher uses two forks, which are picked up in order. A list of integers is used to simulate timeouts for both thinking and eating. The events associated with the philospher's sitting down and getting up have no real interest in this model, so the trivial event `always` is used.

```
philo :: (Fork,Fork)->[Int]->IO (Thread ())
philo (fst, snd) ts =
    trace (events ts)
  where
    events (t1:t2:ts) =
      [ thinking t1,
        sitDown, pickUp fst, pickUp snd,
        eating t2,
        putDown fst, putDown snd, getUp
      ] ++ events ts
    thinking = timeout
    eating   = timeout
    sitDown  = always _
    getUp    = always _
```

A fork ordering is introduced to prevent deadlock. Given a list of n forks $[f_1, .., f_n]$, `ordering` produces a list of pairs with forks (f_i, f_{i+1}) for $1 \leq i < n$, and a special pair (f_1, f_n) instead of (f_n, f_1) to avoid the circular wait.

```
ordering :: [Fork] -> [(Fork,Fork)]
ordering forks = (head forks, last forks)
                 : zip forks (tail forks)
```

The following function initializes the philosopher's dinning. It creates as many forks as needed, establishes the ordering of the forks and then creates the philosophers. Besides the number of philosophers (n), a list with the "random" behaviour for each philosopher is required.

```
dinning :: Int -> [[Int]] -> IO ()
dinning n rs =
  accumulate
    [ fork  | _ <- [1..n]] >>= \forks ->
  sequence
     (zipWith philo (ordering forks) rs)
```

Notice that replacing the philosopher's events by more complex ones, it would be straightforward to implement a graphical interface, for instance, by sending the current state information to a broadcast process which gathers information from all the philosophers.

5 Related Work

Most of the work on concurrency embedded into a functional language rely on the notion of synchronous communications. Thus, these works have been strongly influenced by Hoare's CSP [5] as well as its implementation OCCAM [8]. PFL [6] and AMBER [1] were the first languages which incorporate such concurrency features. Reppy's CML [11] took back those ideas, putting emphasis on abstraction by introducing the notion of event, an implementation of CSP's events. Our work can be viewed as an extension of CML to HASKELL, by integrating the event mechanism into the `IO` monad, and adopting the CSP formalism to define the thread behaviour (traces) using lazy lists. On the other hand, ML-THREADS [2] is yet another concurrency package for ML, providing threads, mutexes, and condition variables.

There are a few papers on concurrency using HASKELL. In [9] the implicit and explicit parallel programming is discussed and a prototype GOFER implementation of synchronous communications is given. In [12], some primitives (quite similar to the primitives exposed in [9]) are used to implement different concurrent formalisms (actors, RPCs, CML, Ada). In [10], a concurrent extension to GLASGOW HASKELL is introduced. That extesion focusses on the efficient implementation of concurrency and aynchronous communication primitives using cells similar to ID's I-var and M-var structures. Neither of them uses infinite lists as traces to model thread behaviour as our work states.

6 Conclusions and Further Work

An extension of the non-strict functional language HASKELL has been exposed. That extension allows the programmer to spawn threads that communicate by different ways (mutexes, write-once variables, and synchronous channels). We have adopted Reppy's higher order concurrency by including the notion of event. Among the combinators included, it must be pointed out the presence of a non-deterministic choice. The greater challange has been to include all these concepts in armony with the non-strict evaluation policy of HASKELL by modifying properly the monadic input/output. In order to model the thread behaviour, lazy lists have been used as Hoare's traces in such a way that it opens the possibility of using functions on list to compose process. Other advanced features of HASKELL such as list comprehension, higher-order functions and, in general, all the expresiveness of purely funcional languages have been shown to represent a great framework for building concurrent abstractions.

A prototype implementation has been developed for two of the platforms used in the distribution of YALE HASKELL and it is freely available. Many examples, including those in [9] as well as interactive applications such as a graphical interface for the dinning philosophers problem, have been implemented. The greater shortcoming of this prototype is the inefficient execution of YALE HASKELL system, an order of magnitud (at least) if compared with GLASGOW HASKELL, although YALE HASKELL is user friendlier because of its interactive toplevel. Besides of inheriting that inefficient execution, the expensive nature of synchronous operations makes our prototype slow. Thus, further work has to be done in order to improve that.

Although the current prototype has been thought to be used on a monoprocessor machine, it seems to be obvious the benefits of extending such a prototype to multiprocessor machines. In particular, we are interested in distributed execution of funcional code and, hence, a concurrency layer might be very useful to implement the underlying model for that transparent distribution. In addition, formal semantics of the given extension should be developed.

References

[1] L. Cardelli. Amber. In G. Cousineau, P.L. Curien, and B. Robinet, editors, *Combinators and functional programming languages*, LNCS 242. Springer Verlag, 1986.

[2] E.C. Cooper and J.G. Morrisett. Adding threads to Standard ML. Technical Report CMU-CS-90-186, Carnagie Mellon University, 1990.

[3] P. Hudak et al. Report on the functional programming language Haskell, Version 1.2. *ACM SIGPLAN Notices*, 27, May 1992.

[4] R. H. Halstead. Multilisp: a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[5] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, Englewood Cliffs, NJ, 1985.

[6] S. Holmstrom. *Polymorphic type systems and concurrent computations in functional languages.* Ph.D. thesis, Chalmers University, Department of Computer Science, 1983.

[7] P. Hudak. Para-Functional Programming in Haskell. In Boleslaw K. Szymanski, editor, *Parallel Functional Languages and Compilers*, Frontier Series, chapter 5, pages 159–196. ACM Press, NY, 1991.

[8] Geraint Jones. *Programming in Occam.* Prentice-Hall international, 87.

[9] M.P. Jones and P. Hudak. Implicit and Explicit Parallel Programming in Haskell. Research Report YALEU/DCS/RR-982, Yale University Department of Computer Science, 1993.

[10] S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Proceedings of the 23rd ACM POPL*, St Petesburg Beach, Florida, 1996.

[11] J.H. Reppy. *Higher-Order Concurrency.* Ph.D. thesis, Cornell University, June 1992.

[12] E. Scholz. Four concurrency primitives for haskell. In *Proceedings of the Haskell Workshop*, pages 1–12, La Jolla, CA, USA, 1995. Yale University Research Report YALEU/DCS/RR-1075.