Programming Reactive Systems in Haskell

Sigbjørn Finne*
Department of Computing Science, University of Glasgow
sof@dcs.gla.ac.uk

Simon Peyton Jones
Department of Computing Science, University of Glasgow
simonpj@dcs.gla.ac.uk

Abstract

Certain classes of applications are naturally described as a network of cooperating components, where each component reacts to input as and when it becomes available. This paper builds on previous work on expressing I/O and state-based algorithms safely in a functional language, and presents a new way of expressing reactive systems in a functional language that does not violate vital semantic properties. The approach taken is a pragmatic one in that it integrates constructs for expressing reactive systems within existing framework for writing I/O bound, state-based programs functionally.

The original contributions of this paper are twofold; first, we show how the existing monadic IO model can be extended to cope with non-determinism, and secondly, we introduce primitives for evaluating I/O actions concurrently.

1 Introduction

A large class of applications are naturally described as a collection of components cooperating to solve some task. Examples include operating systems and graphical user interfaces where partitioning the system into a set of independently running processes or threads of control separates out the different concerns, making for more modular solutions. Writing such reactive systems in a non-strict, purely functional language like Haskell[9] is in its very nature problematical, as the concept of having a component reacting to stimuli from multiple sources cannot be described by a (deterministic) function. Indeed, the straightforward introduction of constructs for expressing such non-determinate behaviour does interfere with fundamental semantic properties of the language [6], disallowing the unrestricted use of equational reasoning.

This paper builds on previous work on expressing I/O and safe state encapsulation, and addresses the problem of how to write reactive systems such

*Supported by a Research Scholarship from the Royal Norwegian Research Council.

as graphical user interface applications or operating systems in a pure functional language. Specifically, support for concurrency and non-determinism is introduced. To make the distinction clear, we use the term *concurrency* to refer to the *explicit* use of separate threads of control to structure a program, and not the *implicit* use of multiple threads or processes to achieve potential performance gains (parallelism). Also, the main reason for introducing non-determinism into a functional setting is the wish to write reactive systems *within* a functional language, rather than a wish to express non-deterministic algorithms.

The original contributions of this paper are:

- *Composable reactive systems.* The approach presented to the expression of reactive systems improves upon existing ways of writing reactive systems such as graphical user interface applications in that it is compositional and concurrent.

  We present our approach through a motivating example in Section 2.

- *Referentially transparent non-determinism.* Non-deterministic operators are provided *without* breaking much valued reasoning properties of the underlying language. This is not only important from the viewpoint of still being able to apply reasoning techniques to functional programs, but also avoids any special treatment of non-deterministic operators during the considerable transformation a program goes through during compilation.

  Our approach to the controlled introduction of non-determinism is presented in Section 3, and we survey some existing approaches to non-determinism in purely functional languages in Section 5.

- *Concurrent actions.* We also introduce a primitive for concurrently executing I/O bound computations or *threads* (Section 2.4). This allows us to partition programs into a set of concurrently running threads, and is used to structure the way we write reactive systems (Section 4.3).

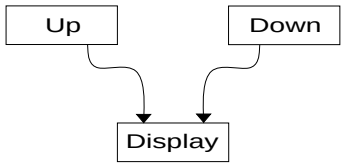
- *Incorporation of non-deterministic operators into I/O framework.* The approach to non-determinism is connected to the IO monad[18], providing a unified framework for constructing programs that engage in I/O and require non-determinism, but still staying within a purely functional setting. This allows the two kinds of programming to be mixed, and offers rich possibilities for extending primitive non-deterministic operators to capture mechanisms such as timeouts, remote procedure calls etc. (Section 4).

2 Motivation

This section introduces our approach to programming reactive systems, motivating it through the discussion of how to best express a simple reactive system.

2.1 Example

As a motivating example, consider a graphical user interface representing a two-way counter with a pair of push buttons and one output display. One button increments the value on the display by one, the other decrements it. The natural way of structuring this application is to partition it into three distinct components:



The components representing the buttons take care of giving them an appearance and appropriate interactive behaviour, while the output display keeps an updated view of the state of the counter. We depict the semantic feedback from the buttons via arcs feeding into the output display, a display that is *shared* between the two buttons.

How do we best translate this abstract view of the counter into a running program? We will leave that question open for the moment, but note that we somehow have to capture the fact that input from the buttons may arrive in any order at the display, and that the components may have to run independently from each other.

This example may appear overly trivial, but the problems it illustrates are found in more complex reactive systems such as an airline reservation system's central database or a resource manager in an operating system.

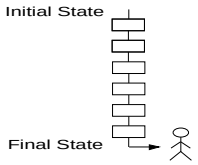
Before looking any further at how to express this application in a functional language, we will briefly review the approach taken to I/O and state-based computations in general.

2.2 IO Programming

The now-standard approach to structuring Haskell programs engaging in I/O is described at length elsewhere [18, 14], but to recap briefly:

- I/O operations side-effect the external world, and are represented by the abstract data type `IO a`. A value of type `IO a` represents an I/O performing *action* that when executed *may* perform side-effects on the external world, before returning a value of type `a` together with an updated world.
- Actions are composed with the sequencing combinator `thenIO :: IO a -> ( a -> IO b) -> IO b`. `m 'thenIO' val -> n` constructs a new IO action, such that when it is performed, `m` is executed first, binding its result to `val` and returning an updated view of the outside world. The action `n` is then executed in this environment.
- The simplest possible I/O operation is `returnIO`, which just returns a value without affecting the external world: `returnIO :: a -> IO a`

In addition, I/O as implemented in the Glasgow Haskell compiler also provides a mechanism for calling C functions through the primitive action `ccall`. Because the only way of composing actions together is with the sequencing combinator `thenIO`, the access to the external world is single-threaded, and it can thus be updated safely in-place, allowing for an efficient compilation of I/O actions (see [18] for details). Informally, a program engaging in I/O may be represented as a thread of primitive actions, strung together with a series of `thenIO`s:



The primitive I/O actions, pictured here as boxes, are joined together in a chain, single-threading the state of the world. To force execution of an I/O bound program, there is a top level *demand* that forces evaluation by demanding the final state produced by the thread. We depict this above by an anthropomorphic character 'pulling' on the thread.

2.3 Explicit interleaving

The basic I/O model, as described in the previous section, for stringing together side-effecting I/O actions is similar to how programs in imperative languages are constructed. One possible way of implementing the two-way counter would be to borrow techniques used in imperative languages.

The conventional approach in such languages is to have a top level dispatcher that fetches events reporting user actions from the window system, and invokes a *callback* function to handle each event. The callback function is selected based on the contents of the event reported, and it makes the required changes to both interface and the state of the application. Using this approach, a set of predefined button and display widgets from a toolkit could be parameterised with appropriate callback functions to implement the two-way counter example. Unfortunately, the whole action-based, event loop paradigm has some serious flaws that does not fit well with functional languages:

- The callback functions deal not only with changes to the appearance of its components, but they also have the responsibility of updating the state of the application. This leads to the sharing of global application state between the different functions, resulting in monolithic, non-modular solutions. Global state could be emulated by threading application state through the different callback functions in addition to the I/O state, as done by Clean's Event I/O system[1].

- The application is centred around the dispatcher or event loop and to ensure adequate responsiveness to the interface, callbacks have to be coded with this in mind. Events have to be serviced every so often; this does not pose any problems for simple examples like a counter, but more complex applications have to be structured as a set of callback functions so that

the dispatcher will be visited at regular intervals. The fact that the application has a graphical user interface dictates the style of programming for the *whole* application, a most unfortunate situation.

Writing the two-way counter this way would certainly be possible, but the solution does not make use of the inherent features of a functional language such as abstraction and the ability to compose entities together with user-defined glue[10]. In the case of the two-way counter, we would very much like to be able to reuse it as part of a larger application, something that is hard to do using this paradigm *without* rewriting it first to fit its context. However, emulating the conventional approach does have the benefit of easily being able to plug into sophisticated toolkits that have been developed around this framework.

A clearer separation between the individual components of the counter can be achieved though by introducing more than one source of demand, each evaluating concurrently. Instead of having to take care of explicitly interleaving execution between the different components using a centralised dispatch mechanism, each component is now responsible for independently servicing input in the form of window system events or input from other components, as and when it becomes available. A response to some input might be to perform I/O operations to change the appearance of the component, e.g. highlighting the button when pressed, or output values to other components.

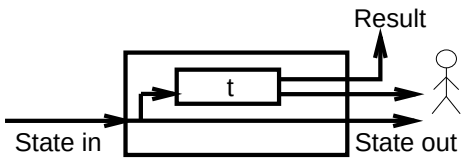
Fortunately, expressing concurrency in a functional language fits well into the I/O framework described in 2.2, and we introduce our approach to concurrency next.

2.4 Concurrent IO actions

IO actions can be executed concurrently using the `forkIO` operator:

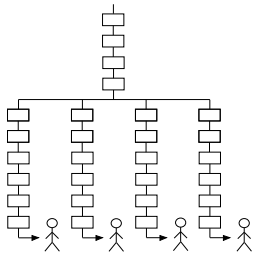
```
forkIO :: IO a -> IO a
```

`forkIO m` dynamically creates a new demand which evaluates the IO action `m` concurrently to the thread that issued the `forkIO` action. The new thread executes in the same context as its creator, so thread creation does not involve any extensive copying of data into a separate address space. Using a plumbing diagram, `forkIO` can be represented as follows:



`forkIO m` splits the state thread into two strands, creating a new demand (the anthropoid) that pulls on the new strand independently of the thread that issued it. The original thread continues executing, but if it tries to inspect the result of `m` before it has been produced, the thread is blocked waiting until the producer thread has produced enough of the result for the blocked thread

to continue.[1] Informally, a set of concurrent I/O threads can be depicted as follows:



`forkIO` splits the state thread into two, and eagerly starts to demand the new strand. Is this 'unsafe', in the sense that is the state of the thread creator visible to the forked off action ? Yes; both threads have access to the 'whole' external world, so if they both choose to write to the same file, say, the resulting state of that file is dependent on the relative scheduling order of the threads. Should we outlaw such indeterminate interactions between the threads by enforcing that the state which they work on don't intersect, using the techniques developed in [14]? No, in most cases this would be overly restrictive, so we place instead a proof obligation on the programmer to ensure that potential interference between threads is benign and intended.[2]

Having got `forkIO`, a thread can now be created for each component in the two-way counter example. Each thread handles input events and, in the case of the buttons, outputs values on a stream to signal that it has been pressed. The resulting structure is now identical to the initial model we gave in Section 2.1, partitioned into a set of interacting components. But exactly how do components interact? We seem to need an operator that once invoked, is able to deliver the next input available from a set of sources. In the case of the output display, such a non-determinate operator is required to be able to service input from the window system and both buttons as values become available on their input streams.

Describe next our approach to incorporating non-determinism safely into the concurrent I/O framework just presented, conclude here by noticing that on the assumption of having an operator for *merging* input from several sources, the running example of a reactive system can be expressed naturally as a set of components communicating, achieving a separation of concerns not present in current ways of writing reactive systems such as graphical user interfaces.

3 Introducing non-determinism

Our idea is simple. We introduce non-deterministic choice through the following primitives:

[1] The mechanism for implicitly blocking threads is just the technique used in parallel functional languages for blocking processes trying to read values that are currently being evaluated.

[2] We also assume that the changes individual threads make to the outside world occur atomically, an assumption that does not hold in general. We return to this issue of atomicity briefly in Section 6.

```
chooseIO ::  (_Data a, _Data b)
         => a -> b -> IO Bool
mergeIO ::  _Data a
        => [a] -> [a] -> IO [a]
```

`chooseIO val_a val_b` is an action that when executed may perform some IO before returning a boolean indicating which of its two arguments it selected together with an updated view of the world. Its operational behaviour is equal to that of `amb`[15], in that:

```
chooseIO  ⊥    val  =>  False
chooseIO  val  ⊥    =>  True
chooseIO  ⊥    ⊥    =>  ⊥
```

where val $\neq \perp$. `chooseIO` diverges if and only if both of its arguments diverge.

`mergeIO` performs a time-based merging of streams, making elements from the input streams available on the output stream when (and if) they appear. The merge is lazy, elements are produced on the result stream on demand.

3.1 Discussion

Introducing non-determinism into a functional language is problematic [6], so why are `chooseIO` and `mergeIO` not harmful? The 'trick' here is to attach the non-deterministic operators to the IO monad, so that the choice is made based on the pair of values it is passed *and* the state of the world. Hence, there is no reason to expect two identical calls to `chooseIO` to pick the same value each time, since the state of the outside world may well be different when the next choice is made. This is similar to an oracular approach [4], appearing to consult the outside world to decide what choice to make. Since we are threading the state of the world through a program, recording the choices made is not necessary, as we can guarantee that actions such as `chooseIO` are not accidentally copied. Thus, a choice will only be made once, so any recording is superflous.

The operators preserve referential transparency for the same reasons as side-effecting programs expressed in an I/O framework do, i.e., the ordering of evaluation is made explicit through threading of state, and only the result value of an action can be shared, not the side-effecting action itself. This prevents indeterminate operations from inadvertently being copied, so `chooseIO` implements *call-time* choice[6], referentially transparent view of non-determinism[19]. Although similar to *oracles* (Section 5), it avoids having to record choices in pseudo-data to catch the cases where a non-deterministic expression is accidentally copied during compilation, say. Making the 'oracle' part of the outside world also avoids the problem of explicitly handling a supply of oracles in your program.

Requiring the non-deterministic operators to be attached to the I/O monad might seem restricting, but this coupling is of practical importance to us, as the main use non-determinism is to express reactive systems such as graphical, I/O bound, user interface applications functionally. Hence, using an operator like `mergeIO`, we can now formulate the two-way counter of Section 2.1 as a set of concurrent I/O threads interacting via streams of values.

However, preserving a property like referential transparency for the overall program does not imply that it is suddenly easy to reason about! The program

is still non-deterministic, and has thus a more complex semantics. But, the linking of non-deterministic choice and the IO monad does have its advantages:

- The choice is separated from its use, making it possible to isolate the non-deterministic parts of a program. Hence, standard reasoning techniques can be applied to the determinate parts.
- Relatively low syntactic burden. There is no danger of the programmer accidentally performing 'plumbing' errors, as the IO monad ensures that a choice is only performed once. This an improvement over the oracle approach (Section 5), where the programmer has to explicitly thread a supply of oracles through the program.
- The choice is pure. This differs from the set-based approach of [11, 22] where the formulation of merge requires the non-deterministic selection between computations that contained a mix of code directly relevant to the selection and code to set up further merging. Because of this mixture, providing an efficient implementation is hard. `chooseIO` avoids this by simply selecting between the two arguments it is passed, leaving it instead up to the context to interpret the boolean it returns.
- A single framework for writing programs that engage in I/O and depend on non-determinism to perform correctly. Previous monadic formulations of non-determinism (Section 5) did not consider how to combine it with state-based monads such as IO.
- Applying the Glasgow Haskell threads extensions technique of using type classes to discriminate against 'non-seqable' expressions, we avoid the problem of handling a choose between partial applications. One practical application of this is that it avoids having to handle the non-deterministic choice between I/O actions:

  ```
  flarp = chooseIO (writeFile fname1 data1)
                   (writeFile fname2 data2)
  ```

  The `flarp` action tries to write to two files, picking the first to complete as its result. If we were to allow a choice between I/O actions, the file operation that 'lost' would now have to be undone. Mechanisms for keeping track of all the changes made by the 'losing' branch are tricky to implement, let alone expensive, so we forbid choice between I/O actions by enforcing membership of the `_Data` type class on arguments to `mergeIO` and chooseIO.

  Membership in the `_Data` type class is taken care of by the compiler, automatically making all non-function values members, avoiding programmer inserted instances to the `_Data` class for every data type defined.

Having presented the non-deterministic primitives, we will now look at examples of how they can be used together with `forkIO`.

4 Examples

4.1 Merging streams

Although we introduced `mergeIO` as a primitive in the previous section, it can be formulated in terms of `chooseIO`:

```
mergeIO :: _Data a => [a] -> [a] -> IO [a]
mergeIO ls rs
 = chooseIO ls rs `thenIO` \ flg ->
   let
    (as,bs) = if flg then
                 (ls,rs)
              else
                 (rs,ls)
   in
    case as of
       []     -> returnIO bs
       (x:xs) -> mergeIO xs bs `thenIO` \ vs ->
                 returnIO (x:vs)
```

`mergeIO` combines two streams of values into one by repeatedly calling `chooseIO` with the stream pair as arguments. The above version of `mergeIO` correctly merges the streams, but it is too inefficient to be the basis for programming reactive systems:

- A call to `chooseIO` result in the concurrent evaluation of the pair of arguments passed in until either of them has been evaluated to WHNF. The computation that 'lost' the choice has to either be undone or allowed to run until completion. If we perform the potentially complex and costly operation of undoing the evaluation of the argument, any further merging is delayed as we cannot proceed until we have restored the previous state of that argument. Allowing the computation to run until completion is also inefficent in the case of `mergeIO`, because subsequent calls to `chooseIO` will create new computations that tries to evaluate the very same value. Although this inefficiency can be worked around, the simplicity of the initial solution is lost, so we choose to provide `mergeIO` as a primitive instead.

- In some cases it is convenient to allow the merge to work ahead of the demand on the merged stream. In the implementation given above, when an element on the result stream is demanded, `chooseIO` will invoked and two concurrent computations will be created to produce the next element on the merged stream. This is inefficient in that the thread creation overheads are incurred for every element demanded. Implementing buffering or avoiding the repeated creation of threads to work on the merged streams is difficult to do in terms of `chooseIO`, which is another why choose to provide `mergeIO` as a primitive.

4.2 Timeout

Assuming that we have `forkIO` of Section 2.4 available, here is how a timeout mechanism can be built:

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

timeoutIO :: _Data a => Int -> a -> IO Bool
timeoutIO period val
 = forkIO (sleeper period) `thenIO` \ unit ->
   chooseIO val unit
```

Timeout is implemented by simply 'listening' to whichever completes first, the passing of the time period or the argument `val`. `forkIO (sleeper ival)` creates a thread that goes to sleep for `period` microseconds before returning `()` to signal the passing of that time period.[3]

This example shows that even though `chooseIO` explicitly forbids choice between IO actions, it is relatively straightforward to create own abstractions on top of the basic choice that selects between *results* of IO actions. It is the responsibility of the programmer to ensure that any side-effects performed by a failed IO action are satisfactorily undone.

4.3 Reactive systems

The example of a two-way counter given in Section 2.1, could be written using `mergeIO` and `forkIO` as follows:[4]

```
button ::  (Int->Int) -> IO [(Int->Int)]

display ::  Int -> [(Int->Int)] -> IO ()
display state (op:ops)
  =
   let
    state' = op state
   in
    ...update output window..
    display state' ops


counter ::  IO ()
counter
  = forkIO (button (+1)) `thenIO`  ups ->
    forkIO (button (-1)) `thenIO`  downs ->
    mergeIO ups downs `thenIO`  mls ->
    display 0 mls
```

[3] `sleeper` is implemented by making a `ccall` to the underlying OS' `sleep` system call, and depends on having non-blocking I/O calls.

[4] This example is only intended to demonstrate how different components are joined up together, and does not reflect how we would like the user to write a graphical user interface application.

The `display` component takes care of the output display window and is passed the merged stream of commands from both buttons. It is simply a stream processor, translating commands into I/O actions that update the output window. The non-deterministic merging of the streams form the buttons is done *externally* to the `display` component. Albeit a small example, but it demonstrates that by separating the different components into separate threads of control, and providing a mechanism for selecting non-determinstically from a number of sources, a much more modular way of expressing reactive systems such as graphical user interface applications is possible. For example, the counter could now be reused as part of other applications *without* having to rewrite it to fit its context. We are currently working on a more general framework for expressing user interfaces using these techniques.

5 Related Work

Several proposals for expressing non-determinism in a functional language have been suggested, we survey some of them here.

One of the simplest ways of introducing non-determinism is with the `amb` operator[15]:

```
amb :: a -> a -> a
```

`amb x y` is locally bottom-avoiding, in that it non-deterministically chooses between `x` or `y` using to the following equations:

```
amb  ⊥  v  =  v
amb  v  ⊥  =  v
amb  ⊥  ⊥  =  ⊥
```

Its operational interpretation is that it evaluates both arguments concurrently until either of them reduces to WHNF. Clearly, `amb` is not a function and its introduction into a lazy functional language destroys important reasoning properties, as substitutivity of expressions is lost [19], the archetypal example of this being:

```
double x = x + x
```

The expression `double (amb 2 1)` is not equal to the instantiated right hand side `(amb 2 1) + (amb 2 1)`, as what was originally one choice has now become two.[5]

`merge`, the stream based version of `amb`, takes a pair of potentially infinite lists and merges them into one, based on the order in which the elements on the streams become available (*fair* merge). Using this operator as a primitive [8, 12] have shown how a variety of functional operating systems can be expressed in a functional framework.

In [4, 2], the potentially non-referentially transparent features of `amb` and Henderson's `merge` are avoided through the introduction of a new data structure, *oracles*. In addition to a pair of values, the `choose` operator[6] is also given

[5] In a lazy language, the expression representing `x` is shared between the right hand side occurrences of `x`, so the above situation will not be a problem. However, situations exist where it is beneficial to break this sharing.

[6] Operationally equivalent to `amb`.

an oracle, an external source which it consults in order to make the 'correct' choice. Internally, an oracle can take on three values, either it indicates what argument choice should select (first or second), or it is marked as being 'undecided'. When `choose` is evaluated, the oracle is consulted, and if it is in an undecided state it is free to select non-deterministically which value to choose, as long the decision made is recorded in the oracle afterwards. Side-effecting of the oracle is safe, as there is no way of seeing different decisions from the same oracle.[7] In the case of the `double` function in the previous section:

```
double (choose o 2 1) = (choose o 2 1) + (choose o 2 1)
```

the first `choose` that is evaluated will record its choice in the oracle, leaving the other `choose` to follow that decision, implementing a singular semantics[6].

Oracles carry the overhead of having to record choices even though each choice is ideally performed only once and programs uding oracles have to explicitly plumb a supply of them through the progrsm, unnecessarily cluttering the program and introducing the possibility of accidental plumbing errors.

The authors of the Fudgets system[5] for writing graphical user interfaces in Haskell, have proposed the adoption of oracles to handle non-determinism when implementing concurrent user interfaces.

An alternative approach to non-determinism is to move it outside the functional language completely. The conventional approach here is to rely on the run-time system provide the actual non-determinism, as exemplified by Stoye[20], Perry [17] and Cupitt[7]. To use Stoye's approach to writing functional operating systems as an example, individual processes are deterministic, modelled as stream processors mapping process requests to resulting responses. Each process is connected up to a global *sorting office*, which merges the output of all the processes, and routes individual messages to the right recipient.[8] The sorting office acts as a global merge, and is implemented as part of the run-time system.

Apart from the obvious restriction on typing of messages in Stoye's approach, which has been solved recently for *static* process networks by [23], all of these approaches rely on having a non-functional run-time system to take care of the 'dirty' bits. Non-deterministic programs have to be modelled as a single collection of processes, each of which is (potentially) a communicating partner with any other.

Related to our main interest in using non-determinism to express user interfaces, this approach is being used by [16] to construct concurrent interactive applications.

The `amb` operator can be turned into a function, by changing its definition just slightly:

```
amb ::  a -> a -> {a}
```

Instead of returning either of its two arguments, the set of possible values `amb` can produce is returned. This idea of representing non-deterministic computations by the sets of values they can denote is used in [11], where a collection of set operators for constructing and composing non-deterministic computations

[7] Assuming, of course, that oracle access is mutually exclusive.

[8] Process addressing is done using a simple numeric scheme.

are given. Referential transparency is retained, as none of the operators provided, inspect or select particular elements contained in a (non-deterministic) set. Non-deterministic choice simply becomes set union:

```
∪ ::  {a} -> {a} -> {a}
```

The set of possible results from a pair of non-deterministic computations is just their combined sum. Computations are composed together using set map:

```
* ::  (a -> b) -> {a} -> {b}
```

When two non-deterministic computations are composed, it is necessary to 'flatten' the resulting set of sets:[9]

```
⋃ ::  {{a}} -> {a}
```

The actual implementation of these sets is done by picking a representative element non-deterministically from the set, which is safe as no operators that inspect individual elements of the sets are provided. As a result, the map operator $*$ is implemented as just function application on the representative element, while $\cup$ non-deterministically selects one of two representative elements.

In [21, 22], the set-based approach is given a monadic formulation, resulting in a less cumbersome and more familiar notation. Here is how merging could be expressed:[10]

```
data L a
unitL  :: a -> L a
bindL  :: L a -> (a -> L b) -> L b
unionL :: L a -> L a -> L a

merge :: [a] -> [a] -> L [a]
merge ls rs
 = (( ls `seq` unitL (ls,rs)) `unionL`
    ( rs `seq` unitL (rs,ls))) `bindL` \ (as,bs) ->
   case as of
        []     -> unitL bs
        (x:xs) -> merge xs bs `bindL` \ ms ->
                  unitL (x:ms)
```

A value of type `(L a)` represents a computation that when evaluated may perform some non-determinate actions before returning a value of type `a`. Non-determinate computations are composed using `bindL`, which has the advantage of automatically giving a handle on values of non-deterministic computations, avoiding the use of the $\bigcup$ operator in [11]. `unionL` selects non-deterministically between two non-determinate computations.

6 Conclusions and Further work

We have presented an approach to writing reactive systems such as graphical user interface applications functionally, where the existing framework for

[9] See [11] for explanations as to why such a flattening operator is required.

[10] `seq :: _Data a => a -> b -> b`, `seq` evaulates its first argument to weak head normal form(WHNF), before returning the second.

describing I/O bound programs was extended to handle concurrency and nondeterminism. These ideas have been implemented on top of the Glasgow Haskell compiler and is currently being used to construct a general framework for creating composable user interfaces in Haskell.

In the previous sections we did not consider the consequences of having concurrency and mutable variables, as introduced in [13]. Briefly, issues of atomicity have to be dealt with here to ensure that mutable variables being shared between threads are accessed exclusively. We are currently experimenting with an approach where mutable variables can have additional synchronisation semantics, in the style of Id's M-structures[3]. Other features being experimented with is the incorporation of mechanisms to handle the other side of synchronisation in concurrent programming, conditional synchronisation. Indeed, having mechanisms for handling both mutual exclusion and conditional synchronisation, an efficient implementation of `mergeIO` can be provided in Haskell, but, sadly, details are outside the scope of this paper.

Acknowledgements

This work was done in the context of the Glasgow Haskell compiler team, and we wish to particularly thank the valiant efforts of Will Partain and Jim Mattson, especially Jim's implementation of the concurrent threads extensions to `ghc` and his speedy responses to numerous questions. Thanks also to the referees for their helpful comments on an earlier version of this paper.

References

[1] Peter Achten and Rinus Pasmeijer. Towards Distributed Interactive Programs in the Functional Programming Language Clean. In *Implementation of Functional Programming Languages Workshop*, University of East-Anglia, Norwich, September 1994.

[2] Lennart Augustsson. Functional non-deterministic programming -or- How to make your own oracle. Technical Report ??, Chalmers University of Technology, March 1989.

[3] Paul S. Barth, Rishiyur S. Nikhil, and Arvind. Non-strict, Functional Language with State. In J. Hughes, editor, *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 538–568. Springer Verlag, August 1991.

[4] F. W. Burton. Nondeterminism with Referential Transparency in Functional Programming Languages. *Computer Journal*, 31(3):243–247, June 1988.

[5] Magnus Carlsson and Thomas Hallgren. FUDGETS – A Graphical User Interface in a Lazy Functional Language. In *ACM Conference on Functional Programming Languages and Computer Architecture*, pages 321 – 330. ACM Press, 1993.

[6] William Clinger. Nondeterministic Call by Need is Neither Lazy Nor by Name. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 226–234, 1982.

[7] John Cupitt. *The Design and Implementation of an Operating System in a Functional Language*. PhD thesis, Department of Computer Science, University of Kent at Canterbury, August 1992.

[8] Peter Henderson. Purely Functional Operating Systems. In J. Darlington, P. Henderson, and D.A. Turner, editors, *Functional Programming and its Applications*, pages 177–192. Cambridge University Press, 1982.

[9] Paul Hudak et al. Report on the Programming Language Haskell Version 1.2. *ACM SIGPLAN Notices*, 27(5), May 1992.

[10] John Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, April 1989.

[11] John Hughes and John O'Donnell. Expressing and Reasoning About Nondeterministic Functional Programs. In Kei Davis and John Hughes, editors, *Proceedings of the Glasgow Functional Workshop*, pages 308–328, 1989.

[12] Simon B. Jones and A. F. Sinclair. Functional Programming and Operating Systems. *Computer Journal*, 32(2):162–174, April 1989.

[13] John Launchbury and Simon L. Peyton Jones. Lazy Functional State Threads. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, June 1994.

[14] John Launchbury and Simon L Peyton Jones. State in Haskell. *Lisp and Symbolic Computation*, 1994. to appear.

[15] John McCarthy. A basic mathematical theory of computation. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, pages 33–70. North-Holland, Amsterdam, 1963.

[16] Rob Noble and Colin Runciman. Concurrent Process Extensions to Gofer. unpublished, September 1994.

[17] Nigel Perry. Towards a Concurrent Object/Process Oriented Functional Language. In *Proceedings of the 15th Australian Compture Science Conference*, 1992.

[18] Simon L. Peyton Jones and Philip Wadler. Imperative Functional Programming. In *ACM Conference on the Principles of Programming Languages*, pages 71 – 84. ACM Press, January 1993.

[19] Harald Sondergaard and Peter Sestoft. Referential Transparency, Definiteness and Unfoldability. *Acta Informatica*, 27:505–517, 1990.

[20] William Stoye. Message-Based Functional Operating Systems. *Science of Computer Programming*, 6:291–311, 1986.

[21] Philip Wadler. Comprehending Monads. In *Proceedings of the ACM SIGPLAN '90 Principles of Programming Languages Conference*. ACM Press, June 1990.

[22] Philip Wadler. The essence of functional programming. In *Proceedings of the ACM SIGPLAN 19th Annual Symposium on Principles of Programming Languages*, January 1992. Invited talk.

[23] Malcolm Wallace and Colin Runciman. Type-checked message-passing between functional processes. In John O'Donnell and Kevin Hammond, editors, *Glasgow Functional Programming Group Workshop*, Ayr, September 1994. Springer Verlag.