Lazy Imperative Programming

John Launchbury
Computing Science Department
Glasgow University
jl@dcs.glasgow.ac.uk

Abstract

In this paper we argue for the importance of lazy state, that is, sequences of imperative (destructive) actions in which the actions are delayed until their results are required. This enables state-based computations to take advantage of the control power of lazy evaluation. We provide some examples of its use, and describe an implementation within Glasgow Haskell.

1 Introduction

There has long been a tension between functional programming languages and their more traditional imperative counterparts. On the one hand, functional languages are commonly more expressive and easier to reason about than imperative languages, but on the other hand, certain algorithms and interactions seem to rely fundamentally on state-based computation. It is clearly worth attempting to combine the strengths of each.

Some languages like Scheme and ML have incorporated imperative actions as side effects. This approach only makes sense in a call-by-value language where the order of evaluation is statically defined. In lazy functional languages like Haskell or Miranda, evaluation order is dynamically determined, so can be immensely difficult to predict. If such languages obtained imperative actions as side-effects of evaluation then serious reasoning difficulties would result. Determining which side effects are to be performed, and worse, in which order, would rely on determining the exact order of evaluation—the very thing that laziness tends to hide! Consequently lazy evaluation and *side effects* do not mix.

There is a way forward however. The recent work of Moggi [Mog89] showed how *monads* could be used to structure denotational semantics, including the handling of state in imperative languages. Wadler recognised that the same principles apply for functional programs and, in particular, showed how the state monad could be used to incorporate imperative operations in a purely functional language [Wad92]. From this work it became clear that lazy evaluation and imperative actions could coexist quite happily, with each being able to pass values to the other.

The Glasgow Haskell compiler incorporates extensions to the lazy functional language Haskell. These extensions implement imperative actions [PW93]. However, the implementation has an important shortcoming: all imperative actions are performed *before* any results are returned. This parallels strict evaluation where all the arguments to a function are evaluated before a result is given, *even if the values of the arguments are not required.* In non-strict languages (as implemented by lazy evaluation, for example) functions may return results without requiring the successful evaluation of arguments whose values are not required.

In this paper we see how to retain the philosophy of non-strict computation, even in the presence of imperative actions. To see the relevance of this, it is worth distinguishing between two distinct classes of state-based computation. Sometimes it is necessary to manipulate the global world state, such as when accessing files or communicating with the outside world, whether as an interactive program or by communicating with other programs. These are *externally state-based* computations. Other times however, computations require purely local state. This may be either for ease of programming, passing a name-supply around for example, or because a given algorithm relies on constant-time update to achieve a given asymptotic complexity. In both cases the state is purely *internal* to the computation, and is utterly invisible to the outside world.

As the state from an internally state-based computation may be discarded at the end of the computation, it makes sense to ask whether *all* the imperative actions should *always* be performed: any that do not affect the final value could be discarded freely. This may be generalised by making the state thread entirely lazy. Hughes argues that the true power of laziness is to decouple control from calculation [Hug89], and we show the same arguments apply to internally state-based computations. We provide some examples of this, and give an implementation of lazy-state in Glasgow Haskell.

The contribution of this paper, therefore, is to show that,

1. lazy state provides a much better fit with lazy evaluation than does strict state; in particular, the usual expressiveness associated with lazy evaluation also applies to internally state-based computations;

2. lazy state may be implemented to provide constant-time access and update: the only penalty compared with strict state is in the creation of extra unevaluated closures—exactly the usual penalty paid by moving from strict to lazy function application.

As this paper builds directly on the work by Peyton Jones and Wadler [PW93], it makes sense to review that first.

2 Imperative Haskell

Imperative features were introduced to Glasgow Haskell for expressing input and output, so the type constructor for state-based computations is called `IO`. A state-based computation that produces a purely functional value of type `a` therefore has type `IO a`. A value of type `IO a` is not a side-effecting computation. Rather it is *a recipe for performing certain actions and returning an appropriate result.* In fact, elements of the IO type are all functions, awaiting an argument representing the world state. When applied to a world state, evaluation of the function produces a result paired with another world state.

By making `IO a` an abstract data type with limited built in operations, it is possible to guarantee that all references to the state are single threaded. Then no state argument ever needs to be passed explicitly: the real external world may be used instead and all updates may be performed in-place.

The main two operations provided on the `IO` type are the familiar monadic combining forms.

```
returnIO :: a -> IO a
thenIO :: IO a -> (a->IO b) -> IO b
```

The former packages up its value into a computation which does nothing except return the value. The latter is a form of function application. It performs the first computation, yielding a value of type `a`. This result is passed to the function argument which, on receiving a value of type `a`, returns a computation designed to produce a value of type `b`. *No value is returned until both arguments to* `thenIO` *have been performed.*

Other `IO` primitives may be built using the `ccall` construct which allows calls to any C function. For example,

```
ccall putchar c
```

is a call to the C function which outputs a character on the standard output. No particular value is returned, so its type is `IO ()`. On the other hand,

```
ccall getchar
```

has type `IO Char`: it is a computation which returns a character value.

Finally, the meaning of a Haskell program is given by a definition of `mainIO`, an identifier of type `IO ()`. To execute a program, `mainIO` is applied to the external world state, returning an updated world state as a result.

3 Delayed Side Effects

An important aspect of the IO monad is that all the imperative actions are forced. Consequently, IO expressions have a very poor match with lazy evaluation. Consider the following expression[1]:

```
fooIO `thenIO` \a->
bazIO `thenIO` \b->
returnIO (a||b)
```

Whenever this expression is performed (applied to the world state) both `fooIO` and `bazIO` are performed, returning (boolean) values bound to `a` and `b` respectively. Only then are these values *or*-ed together to produce the final result. Contrast this with the behaviour of the purely functional expression `foo||baz`. In a non-strict language, `foo` would be evaluated and *only if it returns* `False` *would* `baz` *be evaluated.*

However, as the `IO` type is provided for *external* state computations, this behaviour is quite reasonable. On the main line of IO actions, *every* imperative action has to be performed before the program terminates: the meaning of a program is defined to be the total effect it has on the world. This means that whenever an IO operation is specified, it might as well be performed immediately. There is no benefit in delaying it, since it has to be performed before the program terminates.

On the other hand, the example does provide an argument that expressions which use state purely internally should do so in a lazy way. That is, if we had a way of encapsulating the state in the example above, then `bazIO` should only be performed if `a` is `False`. The result of an encapsulated

[1] The lambda bindings `\a->` and `\b->` are in scope to the end of the expression.

expression should be defined by *the purely functional values it returns, and not by any imperative actions it performs.* If a function uses state to produce a result, and hides that state from everything else (including other invocations of the same function), then there is no reason to perform *all* of the state operations. Rather, *only those which are required by data dependency should be performed.*

There is one important caveat. The order in which imperative actions are performed is important. We must ensure, therefore, that the order in which they are performed remains unchanged; that is, if the value from any action is required, *all earlier actions must first be performed.*

3.1 Passing State Around

Given that the `IO` type is strict in its state actions, we will use a different name for lazy state operations. The only restriction on when such operations are performed is that they must be performed in sequence, so we name the type constructor for such actions `Seq`.

`Seq` is built upon the standard monad of state transformers. A state transformer is a function which, when given a state `s`, produces a pair of results: a value and a new state.

```
type ST s a = s -> (a,s)
```

We provide the standard unit `return` and the sequence combinator called `bind`.

```
return :: a -> ST s a
bind   :: ST s a -> (a->ST s b) -> ST s b

return a s = (a,s)
bind m k s = k a t  where  (a,t) = m s
```

By single-threading the state, the state monad ensures that state operations are sequenced. It does not demand that all state operations are performed. Consider the following example.

```
exampleA :: ST s Bool
exampleA s = someStateOps `bind` \v ->
             return True
```

Because the final value `True` does not depend on the state, the computations in `someStateOps` are not performed. Indeed, not even the initial state is demanded.

If, however, `exampleA` is sequenced together with other code, `exampleB` say, which *does* require the state in order to define the final value, then all the state operations in `someStateOps` will be performed, and in the order originally specified, before any state operations in `exampleB` are performed.

3.2 Discarding State

We define lazy sequences to be state transformers where the state is an abstract data type representing the encapsulated state.

```
type Seq a = ST World a
```

We achieve this encapsulation with the function `newSeq`.

```
newSeq :: Seq a -> a
```

`newSeq` takes a sequence expression, opens up a new, empty imperative context, sequences the imperative computation, and extracts the final value *discarding the final state.*

The evaluation of such a computation mirrors usual lazy evaluation: only those computations and state actions that occur up to the point where the result is sufficiently defined (according to the external demand) are performed; the remainder are suspended. If more of the result is demanded then more computations or imperative actions may be performed until the value is sufficiently defined to satisfy its consumers. As usual, only computations required by data-dependency are performed (recall that imperative actions have the extra data-dependency of a single-threaded state). When, eventually, all references to the final value are discarded, all the remaining actions become garbage, and are never performed.

For the present, we will continue to take it on faith that lazy imperative actions can be performed in a truly imperatively manner (by destructive update), and will turn to examine some examples, showing the sort of behaviour we want to achieve.

3.3 Monad Syntax

For the rest of the paper we use syntactic sugar to refer to monad operations. We extend Haskell expressions with an expression of the form `{Q}` which is expanded as follows[2]:

```
Q ::= E | E;Q | x<-E;Q

{E}      = E
{E;Q}    = E `bind` \_->{Q}
{x<-E;Q} = E `bind` \x->{Q}
```

[2] Haskell actually uses these symbols as layout markers. We will not do so here—every use of {, }, and ; will be for monad syntax.

```
scanLeft :: (a->b->b) -> (b,[a]) -> ([b],b)

scanLeft f (init,xs) = newSeq {v <- newVar init;
                               ys <- scan xs
                                     where    scan   []   =  return []
                                              scan (x:xs) = {val <- readVar v;
                                                             writeVar v (f x val);
                                                             rest <- scan xs;
                                                             return (val:rest)};
                               final <- readVar v;
                               return (ys,final)}
```

Figure 1: Imperative Scan Left

To see this in action, consider the following example:

```
{x <- op1;
 (y,z) <- op2 x;
 op3 z x;
 return y}
```

This is translated into the following expression:

```
op1     `bind`   \x->
op2 x   `bind`   \(y,z)->
op3 z x `bind`   \_->
return y
```

That is, the semicolon is translated into the monad sequence operation `` `bind` ``, and if an explicit pattern is given, it is converted into the pattern on the trailing lambda expression, so causing those names to be in scope across the remainder of the expression. If no pattern is written, then any trailing lambda is given the wildcard pattern which matches everything but binds nothing.

3.4 Scan Left

The first example exhibits the use of an updatable variable. This is an example that is commonly written in a purely functional style, so the value of this example is not about whether to use a state-based computation, but rather to demonstrate that the use of state does not restrict which results may be returned.

For now, we will assume we have the following operations on variables, and treat `Var a` simply as an abstract data type of variables of type `a`.

```
newVar   :: a -> Seq (Var a)
readVar  :: Var a -> Seq a
writeVar :: Var a -> a -> Seq ()
```

Given an initial value `x` say, `newVar x` is a sequence operation which when performed, allocates a variable in the state having initial value `x`, and returns a reference to the variable. Similarly, if `v` is a variable (a state reference), then when `readVar v` is performed (applied to the state) it returns the value of `v` in the state. As may be expected, `writeVar` updates the value of the variable.

Using these we will implement a function `scanLeft`. Given a function, a starting value, and a list, `scanLeft` applies the function repeatedly across the list, working from left to right, and returns a list of partial results, paired with the final answer. For example,

```
scanLeft (+) (0, [1,2,3,4]) = ([0,1,3,6], 10)
```

It is important that the result of `scanLeft` is generated by demand. If only an initial portion of the result list is required, then that is all that should be computed. In particular, even if only some front portion of the input list is defined (and the rest undefined), an equivalent front portion of the result list should still be defined. For example,

```
scanLeft (+) (0, 1:(2:(3:⊥)))
  = (0:(1:(3:⊥)), ⊥)
```

There are two parts to the code given in Figure 1. The first interfaces the main loop with the outside (functional) world. It creates a new imperative context, and within this generates a variable `v` to hold the running total. The main loop of the function is called. This returns the list of partial results and, after reading the final value of `v`, the function returns a pair consisting of the list `ys` and the final result `final`.

The main loop updates the value of the variable `v` for each element of the list, returning a list made up of the value of `v` at the start of the loop followed by the list returned by the rest of the loop.

In a strict imperative framework such as the IO monad (and most imperative languages), no value could be returned until the whole of the list was traversed. Using lazy sequences, however, this is not the case. If only the head of the list is required then very little of the computation is performed: the variable is allocated and initialised, it is read, and the list returned with that value in the head. If even less is required, merely whether the final list is empty for example, then the variable is not even allocated as only `xs` needs to be examined in order to give the structure of `ys`.

As we said above, this example is usually written in a purely functional style. It is important, however, that the same behaviour may be obtained even when state is used: re-expressing the algorithm within the state monad need not change its semantics.

The next example is quite different in that its use of state is *crucial* to obtaining linear efficiency.

3.5 Depth First Search

How can we implement depth first search of a graph efficiently (that is, $0(V+E)$ where V is the number of vertices, and E the number of edges), while still retaining the usual expressiveness and flexibility afforded by lazy functional languages? This problem was the motivating application for this work on lazy imperative actions, and we describe it in some detail here.

3.5.1 Decomposing Graph Algorithms

One major shortcoming of graph algorithms which rely on depth-first search (DFS) is that they are presented dynamically, within the *process of performing* the search. This means that reasoning about the results of the search depend on tracing the computation dynamically. Furthermore, as the code for a particular algorithm is mixed in with the traversal code DFS code is largely unreusable.

The same may be said for many implementations of tree algorithms, but in lazy functional languages it is common to express many tree traversal problems first by a flattening of the tree into a list, and then by traversing the list. The flattening is performed using one of a variety of standard functions (preorder, postorder etc.), and the list processing likewise is often defined in terms of standard components. Complex algorithms can often be expressed as the composition of well-understood simpler components. The intermediate list provides a channel of communication between these standard components. This technique is particularly viable if the list may be consumed as it is produced, otherwise a large intermediate structure is created. The "co-routining" behaviour of lazy evaluation is able to do this.

The same idea is applicable to graphs. Instead of expressing an algorithm *as part of* a depth-first traversal, it is possible to imagine a decomposition similar to that used for many tree algorithms. First, the graph is traversed to produce a depth-first spanning forest (not all the nodes may be reachable from the start node), and secondly the forest is traversed to compute the required information. Again, this is only practicable if the forest is *produced on demand, as it is consumed.*

A forest is a list of general trees, each tree consisting of a node with a value, together with a forest of sub-trees.

```
type Forest a = [Tree a]
data Tree a   = Node a (Forest a)
```

3.5.2 Representing Graphs

There are a number of ways to represent (directed) graphs. Because we want to be explicit about sharing, we represent graphs by adjacency lists. We use a standard Haskell (monolithic) array indexed by vertices, each element being a list of the vertices reachable in one step from that vertex.

One way to build such a graph is from an association list of vertices representing the (directed) edges, together with a pair of verticies indicating the range of valid vertices[3].

```
type Graph = Array Vertex [Vertex]
type Vertex = Int

out :: Graph -> Vertex -> [Vertex]
out g v = g ! v

buildG :: (Vertex,Vertex) ->
          [Assoc Vertex Vertex] -> Graph
buildG is es
  = accumArray (flip (:)) [] is es

verticesG g = [low..high]
  where  (low,high) = bounds g
```

This representation takes a linear amount of space with respect to the size of the graph i.e. the sum of the number of vertices and number of edges. Access to each list of edges takes constant time.

Note that as the graph is represented as a purely functional value, it will not be subject to imperative actions and, in particular, will not be altered in any way by the depth first search. We may pass the same graph around and freely perform many separate searches on it if we wish.

[3] `Vertex` does not have to be `Int` as above, but only needs to be in the Haskell index class `Ix`.

```
dfs    :: Graph -> [Vertex] -> [Tree Vertex]

dfs g vs = newSeq {marks <- newArr (bounds g) False;
                   search vs
                     where  search  []    = return []
                            search (v:vs) = {visited <- readArr marks v;
                                             if visited then
                                                search vs
                                             else
                                               {writeArr marks v True;
                                                as <- search (out g v);
                                                bs <- search vs;
                                                return ((Node v as) : bs)}}
                   }
```

Figure 2: Lazy Depth First Search

3.5.3 Depth First Search

Given the standard von Neumann model, it seems impossible to produce a linear time depth first search without using some element of update-in-place. However, there is only one place in which this is required, namely in setting a mark whenever a particular node has been visited. Often this mark variable is defined to be an extra field in the original graph, but there are a couple of related reasons why this is undesirable. First, the process of performing a search has a side effect on the graph, and this may need to be explicitly removed before another search can commence. Secondly, the component of the algorithm which is necessarily imperative is not identified as such, but rather is mixed up with the rest of the data.

An alternative is to have an array of marks, one for each vertex. At the start of a search we will allocate a new array initialised everywhere to `False`, and make it available for reference and update throughout the search. Once the search is completed the array may be discarded.

The operations on updatable arrays correspond to those on variables:

```
newArr :: (Ix ix) =>
    (ix,ix) -> ele -> Seq (ArrRef ix ele)

readArr :: (Ix ix) =>
    ArrRef ix ele -> ix -> Seq ele

writeArr :: (Ix ix) =>
    ArrRef ix ele -> ix -> ele -> Seq ()
```

The only way a value may be obtained from an array reference is by using the sequence operations. Again we stress that sequence operations are not necessarily strict: the only imperative actions that are performed are those required by data dependency (including retaining the same relative linear order).

The definition of depth first search is given in Figure 2. The function `dfs` is given a list of vertices to search (this is useful for a number of algorithms), and begins with the first. Once the first has been searched, following all the edges from the vertex, and all the descendents of these, etc., the rest of the vertices are explored.

Each (recursive) call of `search` returns a forest. The call `search (out g v)` which is given the edges leading from `v` produces a forest which is built into a tree with `v` at the root—all these nodes are reachable from `v`. The second recursive call (`search vs`) produces a forest of those vertices not reachable from `v` and not previously visited. The tree rooted at `v` is added to the front of this forest giving the complete depth-first forest.

3.5.4 Exploring the Forest

We will give two examples of the use of `dfs`. The first detects the presence of cycles in a graph, and the second identifies strongly-connected components.

Detecting Cycles

Traditionally, in order to find whether a graph contains a cycle, a depth first traversal is performed looking for back edges (edges up to a predecessor in the particular depth first tree). As soon as a back edge is found the search is stopped.

The same effect is obtained using the function `dfs`. We construct a depth-first forest from the graph, and then traverse this forest looking for back edges (we need to refer to the

original graph to obtain these edges). We can express forest traversal by mapping a tree-traversal down the list of trees.

Thus we define a tree traversal function, `treeCycle` say, which takes a graph and a sub-tree of the graph. It traverses the tree and returns `True` as soon as it spots a back edge in the original graph, otherwise it continues to traverse the rest of the tree, and eventually returns `False`.

Assuming this, the cycle-detection program is simply,

```
cycle g = or (map (treeCycle g)
                  (dfs g (verticesG)))
```

As soon as a tree is found for which the graph has a back edge, the `or` function returns `True`, discarding the rest of the forest *which is therefore never produced.* Furthermore, assuming the data dependency of the tree traversal matches that of the DFS generation (left branches first) then only the portion of each DFS tree that is actually traversed will be produced.

Thus we have taken full advantage of lazy-evaluation's ability to provide dynamic control between separate functions: without lazy state it would not have been possible to decouple the traversal of the graph with its mark bits, from the detection of back edges, while still retaining the ability to halt the DFS as soon as a back edge is found.

Strongly-Connected Components

To demonstrate the flexibility of `dfs` we provide a second example of its use, this time the strongly connected components algorithm due to Kosaraju in 1978 (unpublished) [Sha81]. This example does not take particular advantage of laziness, but does show a reuse the DFS code.

For the purposes of this algorithm, we originally specify a graph as a pair of bounds for the vertices (low and high), together with a list of edges. The algorithm performs a depth first search of the graph, generating a forest of vertices. This is flattened to a list (using postorder and then reversing the list) which is used as the seed order for a second depth first search, this time of the reversed graph. Each tree within the resulting depth first forest corresponds to a strongly connected component. We squash each into a list of vertices. The complete implementation is given below[4].

```
scc :: (Vertex,Vertex)       ->
       [Assoc Vertex Vertex] -> [[Vertex]]

scc (low,high) edges
  = (map postorder . dfs g' . reverse . concat
       . map postorder . dfs g) [low..high]
  where
    g  = buildG (low,high) edges
    g' = buildG (low,high) (map switch edges)
    switch (v:=w) = (w:=v)

postorder (Node a ts)
  = concat (map postorder ts) ++ [a]
```

We have seen two examples of using `dfs`. Many other DFS based algorithms can be implemented with similar ease.

4 Implementing Lazy State

Can lazy state be implemented safely and efficiently? By *safely* we mean guaranteeing that the relative ordering of imperative actions remains unchanged, and by *efficiently* we mean using true destructive update to obtain constant time update and access.

It turns out that not only is the answer yes, but that the implementation is surprisingly easy and can be done at source level within Glasgow Haskell. Again we start with a review of the current implementation of `IO`.

4.1 Implementation of IO

So far we have viewed the `IO` type constructor abstractly. Now we see its definition[5].

```
type IO a    = World# -> IORes a
data IORes a = MkIORes a World#
```

Elements of type `IO a` are functions which take a world token (of type `World#`) and return a pair of values, the first of type `a`, the second a new world token.

Here `World#` is an *unboxed data type* [PL91]. The `#` suffix is a lexical convention only with no semantic content, but unboxed types are very different beasts from normal types. In particular, unboxed types have no bottom element, so cannot be undefined. Consequently, any computation involving elements of unboxed values has to produce the value explicitly, *before the next stage of the computation may proceed.* The benefit of using these types is that they expose to the compiler issues both of data representation and of evaluation order, without leaving the purely functional framework.

The fact that the world is unboxed forces `thenIO` to be strict in the world token:

[4] The implementation of `postorder` is not actually linear because of repeated appends, but it may be converted by standard compilation techniques. We have not done so here as it is a little less clear than the naive version, and not relevant to our main interest.

[5] The Glasgow Haskell compiler is currently still under development, so actual names may be subject to changes.

```
returnIO x w# = MkIORes x w#
thenIO m k w# = case (m w#) of
                  MkIORes a v# -> k a v#
```

which in turn forces all the imperative actions specified by `m` to be performed before `k` is evaluated.

The only *truly* primitive IO operation is `ccall` which, once applied to the world token, is implemented by the relevant C call.

There are, of course, some common C accesses which are provided within the standard IO prelude. These include operations for allocating, reading from, and writing to, arrays. We use these in Section 4.4.

4.2 Boxing the World

So far we have left the nature of the world token type `World#` unspecified. In fact, the only part of the compiler which knows the definition on `World#` is the code generator. Everything else views it as an abstract data type (albeit unboxed), representing the total state of the world. Only the code generator takes advantage of the fact that the real external world may be used, and so instantiates `World#` to the one point type.

For sequences, we want a little more information to be contained in the world token. For a start it must be boxed: in the IO monad, the world token is passed around as an unboxed value, and the typechecker ensures that such unboxed arguments are only ever used in strict computations, thus forcing imperative actions in the IO monad to occur immediately. Boxing the world token means that rather than passing around a token (in effect, granting permission to perform an imperative action), we pass pointers to suspended computations which, when evaluated, yield such a token. It is this that allows us to define the lazy sequencer `bind`.

However, we want more than this. The philosophy behind sequences is that they run in their own local state, independent of any other computations, whether state-based or purely functional. We must ensure, therefore, that the sequence operations we provide may only be used in this way. Cross references between supposedly independent state threads must be banned. Otherwise the result of a program could depend on the order of evaluation—the very thing we are trying to avoid.

The best solution to this is almost certainly a stronger type system, perhaps like the effects system [TJ92]. This would determine that no supposedly pure calculation made reference to anything other than its own internal state.

However, in the absence of such an extension, the solution we adopt here is for *each state thread to have its own unique identifier, and for all operations on state-references to check that the reference is being used within the correct world.*

```
data World      = MkWorld WorldToken World#
type WorldToken = Int
```

Whenever an new imperative thread is created, a state token (of type `World`) will be created, having a unique world token number.

4.3 Independent Threads

Implementing an independent thread comes down to implementing `newSeq`. This could be done by defining it as a primitive within the compiler, and arguably that is the correct place. Nonetheless, we can obtain some useful insight by defining it at the "system programmer level" Haskell.

The IO monad provides a (potentially dangerous) primitive value `world#` representing the world. We will use this as the world token for the thread. However, we also need to generate a unique identifier. The problem is well known: it is just *gensym*.

Again, we could implement *gensym* using slightly dirty tricks with the dangerous `performIO`. Interestingly, however, Odersky has recently shown that the lambda calculus may be safely extended with a gensym operator [Ode93]. The term $\nu t.e$ introduces a new name t within e. The only operation provided on names is equality. The resulting calculus is Church-Rosser.

The concrete syntax Odersky proposes for $\nu t.e$ is `new t -> e`. As this provides exactly what we want, we will adopt his syntax.

```
newSeq :: Seq a -> a
newSeq m = fst (new t -> m (MkWorld t world#))
```

4.4 Arrays

In order to implement arrays we use the primitive operations supplied in Glasgow Haskell. The detail of this does not need to be understood merely to appreciate what is going on. The implementation is given in Figure 3.

To perform any array operation, we pattern match against the structure of the state token. As usual the forces the computations defining that value to be performed. Here this has the effect of forcing all previous imperative actions to be performed, as the token is only available after the appropriate state changes have occurred.

```
data ArrRef ix ele = MkArrRef WorldToken (ix,ix) (MutArr# ele)

newArr   :: (Ix ix) => (ix,ix) -> ele ->      Seq (ArrRef ix ele)
readArr  :: (Ix ix) => ArrRef ix ele -> ix ->        Seq ele
writeArr :: (Ix ix) => ArrRef ix ele -> ix -> ele -> Seq ()


newArr ixs@(ix_start, ix_end) init (MkWorld w v#)
  = case ((index ixs ix_end) + 1) of
      MkInt n# -> case (newArr# n# init) v# of
                    MkSeqRMutArr# arr# new# -> (MkArrRef w ixs arr#, MkWorld w new#)

readArr (MkArrRef t ixs arr#) n (MkWorld w v#)
  = worldCheck t w (case index ixs n of
                      MkInt n# -> case (rdArr# n# arr#) v# of
                                    MkSeqR r new# -> (r, MkWorld w new#) )

writeArr (MkArrRef t ixs arr#) n ele (MkWorld w v#)
  = worldCheck t w (case index ixs n of
                      MkInt n# -> case (wrArr# n# arr# ele) v# of
                                    MkSeqR r new# -> ((), MkWorld w new#) )


worldCheck :: WorldToken -> WorldToken -> a -> a
worldCheck t w val | t==w = val
                   | True = error "Illegal State Access"
```

Figure 3: Array Operations

```
type Var a = ArrRef Int a

newVar   :: a -> Seq (Var a)
readVar  :: Var a -> Seq a
writeVar :: Var a -> a -> Seq ()

newVar init    = newArr (0,0) init
readVar v      = readArr v 0
writeVar v val = writeArr v 0 val
```

Figure 4: Variable Operations

In the case of `newArray`, the size of the array is computed using the standard `Ix` class method `index` which maps arbitrary `Ix` types onto the integers. We then call the primitive `IO` operation `newArr#` providing it with the appropriate world token. This returns an unboxed array together with a new world token. The first is packaged up with the `MkArrRef` constructor, and the second with the current world identifier using `MkWorld`.

Notice that the current world identifier is also packaged with the array, so that each array is tagged with an identifier representing the world to which it belongs.

When an array is read, again the state token is forced. Then the world token number built into the array reference is checked to confirm that the array was created within the same world in which it is now being accessed. It is this check (and the corresponding check in `writeArr`) which ensures that references cannot pass between apparently independent threads. If ever any other primitive operations on `Seq` are provided then a similar check should be included.

If the check is successful, then again primitive IO operations are used to read the array, and the result packaged appropriately. Writing to the array is comparable.

Following ML, we implement variables as arrays with a single element. The detail is given in Figure 4.

4.5 Converting IO to Seq

There is a fair degree of mess within the definitions of the array primitives. Much of this can often be encapsulated in the following generic conversion function.

```
cnvIOToSeq :: IO a -> Seq a
cnvIOToSeq m (MkWorld w v#)
  = case (m v#) of
      MkIORes r new# -> (r, MkWorld w new#)
```

A partial application of `cnvIOToSeq` to an IO operation `m` yields a function expecting a value of type `World`. When the results of the operation are required, the `World` value is forced (the pattern matching demands to view the outer constructor). As before, this may provoke a cascade of earlier computations to be performed, many having imperative effects. When at last the `World` token is visible, all previous imperative effects will have been performed. Now the `IO` action `m` is applied to the world token, its imperative effect performed, and an `IO` result is returned. The components of this are extracted, and the new world token is boxed.

5 Conclusion

This paper may be viewed as providing another step in improving the interface between the functional and imperative worlds, as here we allow data dependency to determine which imperative actions are performed and which are delayed[6].

This has two significant advantages: the first is semantic and the second practical. The semantic advantage is a purely functional program may be re-expressed in terms of the state monad without changing its semantics. The practical advantage is that the power of lazy evaluation for decoupling calculation and control may be used even across functions which used internal state.

This paper builds directly on top of the IO work at Glasgow as reported by Peyton Jones and Wadler [PW93], and summarised in Section 2. There are also strong similarities with the λ_{var} work of Odersky, Rabin and Hudak [ORH93], which itself was influenced by Swarup, Reddy and Ireland [SRI91]. Like Peyton Jones and Wadler, λ_{var} provides only for strict imperative actions: no purely functional result is returned until the structure and content of the state is resolved.

One surprising aspect of this work is that it may all be implemented at source level within Glasgow Haskell (albeit at a level typically reserved for systems programming). In particular, no changes to the compiler were required. This provides considerable evidence for the power and flexibility of the built in `IO` monad and unboxed values.

A number of obvious developments present themselves. Already the `Var` type allows all the flexibility of pointers: a value of type `Var (Var Int)` is a pointer to an integer variable, so it makes sense to augment the variables and arrays with unique tags to allow for pointer equality to be defined. Thus the instance of `==` on `Var` would simply compare the tags of the variables and no more.

Sometimes, data-dependency requires that all the imperative actions within a state thread be performed before any result is returned. In this case it makes sense to perform the operations strictly rather than lazily. This corresponds to replacing lazy function application with strict application when the function is provably strict. In this case it is achieved by replacing the `` `bind` `` combinator with a version

[6] However, it is important to recall that when imperative actions are forced, they occur in the same order in which they were specified. There may be room for the clever work found in imperative language implementations whereby imperative actions are reordered according to actual dependency, but that is beyond the scope of the work here. Currently, for example, if after specifying a computation which sets up a mutable array, only one element is read, then all the preceeding imperative actions will be performed, not merely those required to determine the particular element value.

which performs a `case` analysis, rather than constructing a `let` binding. It should be easy to obtain the necessary information from existing strictness analysers.

6 Acknowledgements

I would like to thank Andy Gill, David King, Will Partain, Simon Peyton Jones and Phil Wadler for the discussions we have had about this work. Simon in particular provided many incisive observations which improved both the content and presentation immensely. The approach to graph algorithms briefly described here is the result of joint research with David King.

References

[Hug89] J.Hughes, *Why Functional Programming Matters*, The Computer Journal, Vol 32, No. 2, CUP, April 1989.

[Mog89] E.Moggi, *Computational Lambda Calculus and Monads*, proc. IEEE. Logic in Computer Science, Asilomar, California, 1989.

[ORH93] M.Odersky, D.Rabin and P.Hudak, *Call by Name, Assignment, and the Lambda Calculus*, proc. ACM Principles of Programming Languages 93, Charlston, N.Carolina, 1993.

[Ode93] M.Odersky, *Making Gensym Safe for Functional Programming*, Dept. of Computer Science, Yale Univ, unpublished.

[PL91] S.Peyton Jones and J.Launchbury, *Unboxed Values as First Class Citizens*, proc. ACM Functional Programming Languages and Computer Architecture, Boston, LNCS 523, S-V, 1991.

[PW93] S.Peyton Jones and P.Wadler, *Imperative Functional Programming*, proc. ACM Principles of Programming Languages 93, Charlston, N.Carolina, 1993.

[Sha81] Sharir, *A Strong Connectivity Algorithm & its Application in Data Flow Analysis*, in Computers and Mathematics with Applications 7:1, pp. 67-72, 1981.

[SRI91] V.Swarup, U.Reddy and E.Ireland, *Assignments for Applicative Languages*, proc. FPCA 91, LNCS 523, London, 1991.

[TJ92] J-P.Talpin and P.Jouvelot, *Polymorphic type, region and effect inference*, Jounal of Functional Programming, Vol 2, Part 3, CUP, July 1992.

[Wad92] P.Wadler, *Comprehending Monads*, MSCS, vol 2, pp 461-493, CUP, 1992.