Lazy Functional State Threads

John Launchbury and Simon L Peyton Jones
University of Glasgow
Email: {simonpj,jl}@dcs.glasgow.ac.uk. Phone: +44-41-330-4500

March 10, 1994

Abstract

Some algorithms make critical internal use of updatable state, even though their external specification is purely functional. Based on earlier work on monads, we present a way of securely encapsulating stateful computations that manipulate multiple, named, mutable objects, in the context of a non-strict, purely-functional language.

The security of the encapsulation is assured by the type system, using parametricity. Intriguingly, this parametricity requires the provision of a (single) constant with a rank-2 polymorphic type.

A shorter version of this paper appears in the Proceedings of the ACM Conference on Programming Languages Design and Implementation (PLDI), Orlando, June 1994.

1 Introduction

Purely functional programming languages allow many algorithms to be expressed very concisely, but there are a few algorithms in which in-place updatable state seems to play a crucial role. For these algorithms, purely-functional languages, which lack updatable state, appear to be inherently inefficient (Ponder, McGeer & Ng [1988]).

Take, for example, algorithms based on the use of incrementally-modified hash tables, where lookups are interleaved with the insertion of new items. Similarly, the union/find algorithm relies for its efficiency on the set representations being simplified each time the structure is examined. Likewise, many graph algorithms require a dynamically changing structure in which sharing is explicit, so that changes are visible non-locally.

There is, furthermore, one absolutely unavoidable use of state in every functional program: input/output. The plain fact of the matter is that the whole purpose of running a program, functional or otherwise, is to make some side effect on the world — an update-in-place, if you please. In many programs these I/O effects are rather complex, involving interleaved reads from and writes to the world state.

We use the term "stateful" to describe computations or algorithms in which the programmer really does want to manipulate (updatable) state. What has been lacking until now is a clean way of describing such algorithms in a functional language — especially a non-strict one — without throwing away the main virtues of functional languages: independence of order of evaluation (the Church-Rosser property), referential transparency, non-strict semantics, and so on.

In this paper we describe a way to express stateful algorithms in non-strict, purely-functional languages. The approach is a development of our earlier work on monadic I/O and state encapsulation (Launchbury [1993]; Peyton Jones & Wadler [1993]), but with an important technical innovation: we use parametric polymorphism to achieve safe encapsulation of state. It turns out that this allows mutable objects to be named without losing safety, and it also allows input/output to be smoothly integrated with other state mainpulation.

The other important feature of this paper is that it describes a complete system, and one that is implemented in the Glasgow Haskell compiler and freely available. The system has the following properties:

- Complete referential transparency is maintained. At first it is not clear what this statement means: how can a stateful computation be said to be referentially transparent? To be more precise, a stateful computation is a *state transformer*, that is, a function from an initial state to a final state. It is like a "script", detailing the actions to be performed on its input state. Like any other function, it is quite possible to apply a single stateful computation to more than one input state.

  So, a state transformer is a pure function. But, because we guarantee that the state is used in a single-threaded way, the final state can be constructed by modifying the input state *in-place*. This efficient implementation respects the purely-functional semantics of the state-transformer function, so all the usual techniques for reasoning about functional programs continue to work. Similarly, stateful programs can be exposed to the full range of program transformations applied by a compiler, with no special cases or side conditions.

- The programmer has complete control over where in-place updates are used and where they are not. For example, there is no complex analysis to determine when an array is used in a single-threaded way. Since the viability of the entire program may be predicated on the use of in-place updates, the programmer must be confident in, and be able to reason about, the outcome.

- Mutable objects can be *named*. This ability sounds innocuous enough, but once an object can be named its use cannot be controlled as readily. Yet naming is important. For example, it gives us the ability to manipulate multiple mutable objects simultaneously.

- Input/output takes its place as a specialised form of stateful computation. Indeed, the type of I/O-performing computations is an instance of the (more polymorphic) type of stateful computations. Along with I/O comes the ability to call imperative procedures written in other languages.

- It is possible to *encapsulate* stateful computations so that they appear to the rest of the program as pure (stateless) functions which are *guaranteed* by the type system to have no interactions whatever with other computations, whether stateful or otherwise (except via the values of arguments and results, of course).

  Complete safety is maintained by this encapsulation. A program may contain an arbitrary number of stateful sub-computations, each simultaneously active, without concern that a mutable object from one might be mutated by another.

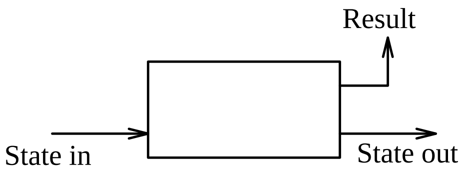
- Stateful computations can even be performed *lazily* without losing safety. For example, suppose that stateful depth-first search of a graph returns a list of vertices in depth-first order. If the consumer of this list only evaluates the first few elements of the list, then only enough of the stateful computation is executed to produce those elements.

2 Overview

This section introduces the key ideas of our approach to stateful computation. We begin with the programmer's-eye-view.

2.1 State transformers

A value of type `(ST s a)` is a computation which transforms a state indexed by type `s`, and delivers a value of type `a`. You can think of it as a box, like this:

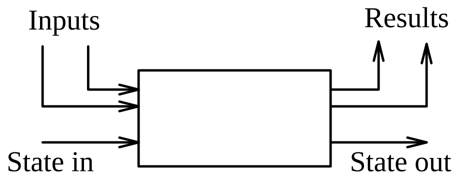


Notice that this is a purely-functional account of state. The "`ST`" stands for "a state transformer", which we take to be synonymous with "a stateful computation": the computation is seen as transforming one state into another. (Of course, it is our intention that the new state will actually be constructed by modifying the old one in place, a matter to which we return in Section 6.) A state transformer is a first-class value: it can be passed to a function, returned as a result, stored in a data structure, duplicated freely, and so on.

A state transformer can have other inputs besides the state; if so, it will have a functional type. It can also have many results, by returning them in a tuple. For example, a state transformer with two inputs of type `Int`, and two results of type `Int` and `Bool`, would have the type:

```
Int -> Int -> ST s (Int,Bool)
```

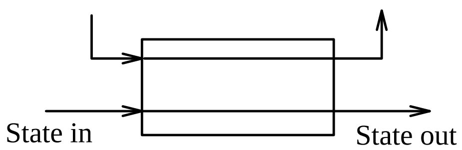
Its picture might look like this:



The simplest state transformer, `returnST`, simply delivers a value without affecting the state at all:

```
returnST :: a -> ST s a
```

The picture for `returnST` is like this:



2.2 References

What, then, is a "state"? Part of every state is a finite mapping from *references* to values. (A state may also have other components, as we will see in Section 4.) A reference can be thought of as the name of (or address of) a *variable*, an updatable location in the state capable of holding a value. The following primitive operations are provided:

```
newVar   :: a -> ST s (MutVar s a)
readVar  :: MutVar s a -> ST s a
writeVar :: MutVar s a -> a -> ST s ()
```

The function `newVar` takes an initial value, of type `a`, say, and delivers a state transformer of type `ST s (MutVar s a)`. When this is applied to a state, it allocates a fresh reference — that is, one currently not used in the state. It augments the state with a mapping from this reference to the supplied value, and returns the reference along with the modified state.

The type `MutVar s a` is the type of references allocated from a store of type `s`, containing a value of type `a`. Notice that, unlike SML's `Ref` types, for example, `MutVar`s are parameterised over the type of the state as well as over the type of the value to which the reference is mapped by the state. (We use the name `MutVar` for the type of references, rather than `Ref`, specifically to avoid confusion with SML.)

Given a reference `v`, `readVar v` is a state transformer which leaves the state unchanged, but uses the state to map the reference to its value.

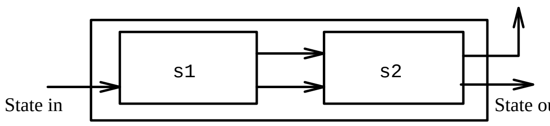
The function `writeVar` transforms the state so that it maps the given reference to a new value. Notice that the reference itself *does not change*; it is the *state* which is modified. `writeVar` delivers a result of the unit type `()`, a type which only has one value (apart from bottom), also written `()`. A state transformer of type `ST s ()` is useful only for its effect on the state.

2.3 Composing state transformers

State transformers can be composed in sequence, to form a larger state transformer, using `thenST`, which has type

```
thenST :: ST s a -> (a -> ST s b) -> ST s b
```

The picture for `(s1 ‘thenST‘ s2)` is like this[1]:



Notice that the two computations must manipulate state indexed by the same type, `s`. Notice also that `thenST` is inherently sequential, because the state consumed by the second computation is that produced by the first. Indeed, we often refer to a state transformer as a *thread*, invoking the picture of a series of primitive stateful operations "threaded together" by a state passed from one to the next.

[1] Backquotes are Haskell's notation for an infix operator.

Putting together what we have so far, here is a "procedure" which swaps the contents of two variables:

```
swap :: MutVar s a -> MutVar s a -> ST s ()
swap v w = readVar v          ‘thenST‘ (\a ->
           readVar w          ‘thenST‘ (\b ->
           writeVar v b       ‘thenST‘ (\_ ->
           writeVar w a)))
```

The syntax needs a little explanation. The form "`\a->e`" is Haskell's syntax for a lambda abstraction. The body of the lambda abstraction, `e`, extends as far to the right as possible. So in the code for `swap`, the second argument of the first `thenST` extends all the way from the `\a` to the end of the function. That's just as you would expect: the second argument of a `thenST` is meant to be a function. The "`_`" in the second-last line is a wild-card pattern, which matches any value. We use it here because the `writeVar` does not return a value of interest.

The parentheses can be omitted, since infix operations bind less tightly than the lambda abstraction operator. Furthermore, we provide a special form of `thenST`, called `thenST_`, with the following type signature:

```
thenST_ :: ST s () -> ST s b -> ST s b
```

Unlike `thenST` its second argument is not a function, so the lambda isn't required. So we can rewite `swap` as follows:

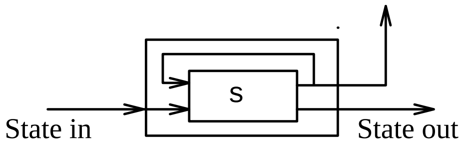
```
swap :: MutVar s a -> MutVar s a -> ST s ()
swap v w = readVar v          ‘thenST‘ \a ->
           readVar w          ‘thenST‘ \b ->
           writeVar v b       ‘thenST_‘
           writeVar w a
```

When `swap v w` is executed in a state thread (that is, when given a state), `v` is dereferenced, returning a value which is bound to `a`. Similarly the value of `w` is bound to `b`. New values are then written into the state at these locations, these values being `b` and `a` respectively.

In addition to `thenST` and `returnST`, we have found it useful to introduce one other "plumbing" combinator, `fixST`. It has the type

```
fixST :: (a -> ST s a) -> ST s a
```

and the usual knot-tying semantics, which we depict thus:



This is the only point that relies on laziness. Everything else in the paper is directly applicable to strict languages.

2.4 Encapsulation

So far we have been able to combine state transformers to make larger state transformers, but how can we make a state transformer part of a larger program which does not manipulate state at all? What we need is a function, `runST`, with a type something like the following:

```
runST :: ST s a -> a
```

The idea is that `runST` takes a state transformer as its argument, conjures up an initial empty state, applies the state transformer to it, and returns the result while discarding the final state. The initial state is "empty" in the sense that no references have been allocated in it by `newVar`; it is the empty mapping.

But there seems to be a terrible flaw: what is to prevent a reference from one thread being used in another? For example:

```
let v = runST (newVar True)
in
runST (readVar v)
```

Here, the reference allocated in the first `runST`'s thread is used inside the second `runST`. Doing so would be a great mistake, because reads in one thread are not sequenced with respect to writes in the other, and hence the result of the program would depend on the evaluation order used to execute it. It seems at first that a runtime check might be required to ensure that references are only dereferenced in the thread which allocated them. Unfortunately this would be expensive. Even worse, our experience suggests that it is surprisingly tricky to implement such a check — the obvious ideas fail as it then becomes possible to test the *identity* of a thread so losing referential transparency — and we still do not know a straightforward way to do so.

This problem brings us to the main technical contribution of the paper: the difficulties with `runST` can all be solved by giving it a more specific type. The type given for `runST` above is implicitly universally quantified over both `s` and `a`. If we put in the quantification explicitly, the type might be written:

```
runST :: ∀s,a. (ST s a -> a)
```

Now, what we *really* want to say is that `runST` should only be applied to a state transformer which uses `newVar` to create any references which are used in that thread. To put it another way, the argument of `runST` should not make any assumptions about what has already been allocated in the initial state. That is, `runST` *should work regardless of what initial state it is given.* So the type of `runST` should be:

```
runST :: ∀a. (∀s. ST s a) -> a
```

This is not a Hindley-Milner type, because the quantifiers are not all at the top level; it is an example of rank-2 polymorphism (McCracken [1984]).

Why does this type prevent the "capture" of references from one thread into another? Consider our example again

```
let v = runST (newVar True)
in
runST (readVar v)
```

In the last line a reference `v` is used in a stateful thread `(readVar v)`, even though the latter is supposedly encapsulated by `runST`. This is where the type checker comes into its own. During typechecking, the type of `readVar v` will depend on the type of `v` so, for example, the type derivation will contain a judgement of the form:

$$\{\ldots, \texttt{v} : \texttt{MutVar s Bool}\} \vdash \texttt{readVar v} : \texttt{ST s Bool}$$

Now in order to apply `runST` we have to be able to generalise the type of `readVar v` with respect to `s`, but we cannot as `s` is free in the type environment: `readVar v` simply does not have type `∀s.ST s Bool`.

What about the other way round? Let's check that the type of `runST` prevents the "escape" of references from a thread. Consider the definition of `v` above:

```
v = runST (newVar True)
```

Here, `v` is a reference that is allocated within the thread, but then released to the outside world. Again, consider what happens during typechecking. The expression `(newVar True)` has type `ST s (MutVar s Bool)`, which will generalise nicely to `∀s.ST s (MutVar s Bool)`. However, this still does not match the type of `runST`. To see this, consider the instance of `runST` with `a` instantiated to `MutVar s Bool`:

```
runST  ::  (∀s'. ST s' (MutVar s Bool))
           -> MutVar s Bool
```

We have had to rename the bound variable `s` in the type of `runST` to avoid it erroneously capturing the `s` in the type `MutVar s Bool`. The argument type now doesn't match `v`'s type. Indeed there is no instance of `runST` which can be applied to `v`.

Just to demonstrate that the type of `runST` does allow some nice examples here is one that is fine:

```
f :: MutVar s a -> MutVar s a
f v = runST (newVar v `thenST` \w->
             readVar w)
```

where `v` is a reference from some arbitrary state thread. Because `v` is not accessed, its state type does not affect the local state type of the short thread (which is in fact totally polymorphic in `v`). Thus it is fine for an encapsulated state thread to manipulate references from other threads so long as no attempt is made to dereference them.

In short, by the expedient of giving `runST` a rank-2 polymorphic type we can enforce the safe encapsulation of state transformers. More details on this are given in Section 5.2, where we show that `runST`'s type can be accommodated with only a minor enhancement to the type checker.

3 Array references

So far we have introduced the idea of references (Section 2.2), which can be thought of as a single mutable "box". Sometimes, though we want to update an array which should be thought of as many "boxes", each independently mutable. For that we provide primitives to allocate, read and write elements of arrays. They have the following types[2]:

```
newArr    :: Ix i => (i,i) -> elt
                  -> ST s (MutArr s i elt)
readArr   :: Ix i => MutArr s i elt -> i
                  -> ST s elt
writeArr  :: Ix i => MutArr s i elt -> i -> elt
                  -> ST s ()
freezeArr :: Ix i => MutArr s i elt
                  -> ST s (Array i elt)
```

Like references, `newArr` allocates a new array whose bounds are given by its first argument. The second argument is a value to which each location is initialised. The state transformer returns a reference to the array, which we call an *array reference*. The functions `readArr` and `writeArr` do what their names suggest. The result is undefined if the index is out of bounds.

The interesting function is `freezeArr` which turns a `MutArr` into a standard Haskell array. The latter is an immutable value, which can certainly be returned from a stateful thread, and hence lacks the parameterisation on the state `s`. Operationally speaking, `freezeArr` takes the name of an array as its argument, looks it up in the state, and returns a copy of what it finds, along with the unaltered state. The copy is required in case a subsequent `writeArr` changes the value of the array in the state, but it is sometimes possible to avoid the overhead of making the copy (see Section 6.2.3).

We give two examples of mutable arrays in action, but leave the larger one to the Appendix.

3.1 Haskell Arrays

Using mutable arrays, we shall define the Haskell "primitive" `accumArray`, a high level array operation with the type[3]:

```
accumArray :: Ix i => (a->b->a) -> a -> (i,i)
                    -> [(i,b)] -> Array i a
```

The result of a call `(accumArray f x bnds ivs)` is an array whose size is determined by `bnds`, and whose values are defined by separating all the values in the list `ivs` according to their index, and then performing a left-fold operation, using `f`, on each collection, starting with the value `x`.

Typical uses of `accumArray` might be a histogram, for example:

```
hist :: Ix i => (i,i) -> [i] -> Array i Int
hist bnds is = accumArray (+) 0 bnds
                     [(i,1)|i<-is, inRange bnds i]
```

which counts the occurrences of each element of the list `is` that falls within the range given by the bounds `bnds`. Another example is bin sort:

```
binSort :: Ix i => (i,i) -> (a->i)
                -> [a] -> Array i a
binSort bnds key vs
 = accumArray (flip(:)) [] bnds [(key v,v)|v<-vs]
```

where the value in `vs` are placed in bins according to their key value as defined by the function `key` (whose results are assumed to lie in the range specified by the bounds `bnds`). Each bin — that is, each element of the array — will contain a list of the values with the same key value. The lists start empty, and new elements are added using a version of cons in which the order of arguments is reversed. In both examples, the array is built by a single pass along the input list.

The implementation of `accumArray` is as follows.

```
accumArray bnds f z ivs = runST
   (newArr bnds z  `thenST` \a ->
    fill a f ivs   `thenST_`
    freezeArr a)

fill a f [] = returnST ()
fill a f ((i,v):ivs)
 = readArr a i          `thenST` \x ->
   writeArr a i (f x v)  `thenST_`
```

[2] The "`Ix i =>`" part of the type is just Haskell's way of saying that the type `a` must be an index type; that is, there must be a mapping of a value of type `a` to an offset in a linear array. Integers, characters and tuples are automatically in the `Ix` class, but array indexing is not restricted to these. Any type for which a mapping to `Int` is provided (via an `instance` declaration for the class `Ix` at that type) will do.

[3] Technically the `(i,b)` should be `Assoc i b`

```
    fill a f ivs)
```

On evaluating a call to `accumArray`, a new state thread is generated. Within this thread an array is allocated, each element of which is initialised to `z`. The reference to the array is named `a`. This is passed to the `fill` procedure, together with the accumulator function `f`, and the list of index/value pairs.

When this list is exhausted, `fill` simply returns. If there is at least one element in the list, it will be a pair `(i,v)`. The array `a` is accessed at location `i`, the value obtained being bound to `x`, and a new value, namely `(f x v)`, is written into the array, again at location `i`. Then `fill` is called recursively on the rest of the list.

Once `fill` has finished, the array is frozen into an immutable Haskell array which is returned from the thread.

Using mutable-array operations has enabled us to describe a complex array "primitive" in terms of much simpler operations. Not only does this make the compiler-writer's job easier, but it also allows programmers to define their own variants for, say, the cases when `accumArray` does not match their application precisely.

The example is also interesting because of its use of encapsulated state. The *implementation* (or internal details) of `accumArray` is imperative, but its *external behaviour* is purely functional. Even the presence of the state cannot be detected from outside the definition of `accumArray`.

3.1.1 Combining State Transformers

Because state transformers are first class values, we can use the power of the functional language to define new combining forms. One that would be useful in the example above is for sequencing a list of "procedures":

```
seqST :: [ST s ()] -> ST s ()
seqST = foldr thenST_ (returnST ())
```

Using this the example above can be rewritten:

```
accumArray bnds f z ivs = runST
  (newArr bnds z                `thenST` \a ->
   seqST (map (update a f) ivs) `thenST_`
   freezeArr a)

update a f (i,v) = readArr a i `thenST` \x->
                   writeArr a i (f x v)
```

The local function `update` takes an index/value pair and evaluates to a state transformer which updates the array referenced by `a`. Mapping this function down the list of index/value pairs `ivs` produces a list of state transformers, and these are sequenced together by `seqST`.

4 Input/output

Now that we have the state-transformer framework in place, we can give a new account of input/output. An I/O-performing computation is of type `ST RealWorld a`; that is, it is a state transformer transforming a state of type `RealWorld`, and delivering a value of type `a`. The only thing which makes it special is the type of the state it transforms, an abstract type whose values represent the real world. It is convenient to use a type synonym to express this specialisation:

```
type IO a = ST RealWorld a
```

Since `IO a` is an instance of `ST s a`, it follows that all the state-transformer primitives concerning references and arrays work equally well when mixed with I/O operations. More than that, the same "plumbing" combinators, `thenST`, `returnST` and so on, work for I/O as for other state transformers. In addition, however, we provide a variety of I/O operations that work only on the IO instance of state (that is, they are *not* polymorphic in the state), such as:

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

It is easy to build more sophisticated I/O operations on top of these. For example:

```
putString :: [Char] -> IO ()
putString   []   = returnST ()
putString (c:cs) = putChar c  `thenST_`
                   putString cs
```

or, equivalently,

```
putString cs = seqST (map putChar cs)
```

There is no way for a caller to tell whether `putString` is "primitive" or "programmed". Indeed, `putChar` and `getChar` are not primitive either. There is actually only one primitive I/O operation, called `ccall`, which allows the Haskell programmer to call any C procedure. For example, `putChar` is defined like this:

```
putChar :: Char -> IO ()
putChar c = ccall putchar c  `thenST` \_ ->
            returnST ()
```

That is, the state transformer `(putChar c)` transforms the real world by calling the C function `putchar`, passing it the character `c`. The value returned by the call is ignored, as indicated by the "_" wild card. Similarly, `getChar` is implemented like this:

```
getChar :: IO Char
getChar = ccall getchar
```

`ccall` is actually implemented as a new language construct, rather than as an ordinary function, because we want it to work regardless of the number and type of its arguments. The restrictions placed on its use are:

- All the arguments, and the result, must be types which C understands: `Int`, `Float`, `Double`, `Bool`, or `Array`. There is no automatic conversion of more complex structured types, such as lists or trees.
- The first "argument" of `ccall`, which is the name of the C function to be called, must appear literally. It is really part of the construct.

4.1 Running IO

The `IO` type is a particular instance of state transformers so, in particular, I/O operations are not polymorphic in the state. An immediate consequence of this is that *IO operations cannot be encapsulated using* `runST`. Why not? Again, because of `runsST`'s type. It demands that its state transformer argument be universally quantified over the state, but that is exactly what `IO` is not!

Fortunately, this is exactly what we want. If IO operations could be encapsulated then it would be possible to write apparently pure functions, but whose behaviour depended on external factors, the contents of a file, user input, a shared C variable etc. The language would no longer exhibit referential transparency.

However, this does leave us with a problem: how are IO operations executed? The answer is to provide a top level identifier,

```
mainIO :: IO ()
```

and to define the meaning of a program in terms of it. When a program is executed, `mainIO` is applied to the true external world state, and the meaning of the program is given by the final world state returned by the program (including, of course, all the incremental changes en route).

By this means it is possible to give a full definition of Haskell's standard input/output behaviour (involving lists of requests and responses) as well as much more. Indeed, the Glasgow implementation of the Haskell I/O system is itself now written entirely in Haskell, using `ccall` to invoke Unix I/O primitives directly. The same techniques have been used to write libraries of routines for calling X, etc.

5 Formal semantics

Having given the programmer's eye view, it is time now to be more formal. In this section we present the static and dynamic semantics of a small language supporting state transformers, and give an outline proof of safety for the encapsulation.

Up to now, we have presented state transformers in the context of the full-sized programming language Haskell, since that is where we have implemented the ideas. Here, however, it is convenient to restrict ourselves to a smaller language which includes only the essentials.

5.1 A Language

We focus on lambda calculus extended with the state transformer operations. The syntax of the language is given by:

$$\begin{array}{lcl} e & ::= & x \ \mid\ k \ \mid\ e_1\ e_2 \ \mid\ \lambda x.e \ \mid \\ & & \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \ \mid\ \texttt{runST}\ e \ \mid \\ & & \texttt{ccall}\ x\ e_1 \cdots e_n \end{array}$$

$$\begin{array}{lcl} k & ::= & \ldots \ \mid\ \texttt{thenST} \ \mid\ \texttt{returnST} \ \mid\ \texttt{fixST} \ \mid \\ & & \texttt{newVar} \ \mid\ \texttt{readVar} \ \mid\ \texttt{writeVar} \ \mid \\ & & \texttt{newArr} \ \mid\ \texttt{readArr} \ \mid\ \texttt{writeArr} \ \mid \\ & & \texttt{freezeArr} \end{array}$$

5.2 Types

Most of the type rules are the usual Hindley-Milner rules. The most interesting addition is the typing judgement for `runST`. Treating it as a language construct avoids the need to go beyond Hindley-Milner types. So rather than actually give `runST` the type

```
runST :: ∀a.(∀s.ST s a) -> a
```

as suggested in the introduction, we ensure that its typing judgment has the same effect. So because it is consistent with the rank-2 type, our previous intuition still applies.

As usual, we talk both of *types* and *type schemes* (that is, types possibly with universal quantifiers on the outside). We use T for types, S for type schemes, and K for type constants such as Int and $Bool$. In addition we use C to range over the subset of K that correspond to the "C-types" described in Section 4.

$$\begin{array}{lcl} T & ::= & t \ \mid\ K \ \mid\ T_1 \rightarrow T_2 \ \mid\ \texttt{ST}\ T_1\ T_2 \ \mid \\ & & \texttt{MutVar}\ T_1\ T_2 \ \mid\ \texttt{MutArr}\ T_1\ T_2 \end{array}$$

$$S \ ::= \ T \ \mid\ \forall t.S$$

Note that the `MutArr` type constructor has only two arguments here. The missing one is the index type. For the purposes of the semantics we shall assume that arrays are always indexed by naturals, starting at 0. The type rules are given in Figure 1. Γ ranges over type environments (that is, partial functions from references to types), and we write $FV(T)$ for the free variables of type T and likewise for type environments.

$$\mathit{APP}\qquad \frac{\Gamma \vdash e_1 : T_1 \rightarrow T_2 \qquad \Gamma \vdash e_2 : T_1}{\Gamma \vdash (e_1\ e_2) : T_2}$$

$$\mathit{LAM}\qquad \frac{\Gamma, x : T_1 \vdash e : T_2}{\Gamma \vdash \lambda x.e : T_1 \rightarrow T_2}$$

$$\mathit{LET}\qquad \frac{\Gamma \vdash e_1 : S \qquad \Gamma, x : S \vdash e_2 : T}{\Gamma \vdash (\texttt{let}\ x = e_1\ \texttt{in}\ e_2) : T}$$

$$\mathit{VAR}\qquad \Gamma, x : S \vdash x : S$$

$$\mathit{SPEC}\qquad \frac{\Gamma \vdash e : \forall t.S}{\Gamma \vdash e : S[T/t]} \quad t \notin FV(T)$$

$$\mathit{GEN}\qquad \frac{\Gamma \vdash e : S}{\Gamma \vdash e : \forall t.S} \quad t \notin FV(\Gamma)$$

$$\mathit{CCALL}\qquad \frac{\Gamma \vdash e_1 : C_1 \ \cdots \ \Gamma \vdash e_n : C_n}{\Gamma \vdash (\texttt{ccall}\ x\ e_1 \ldots e_n) : C}$$

$$\mathit{RUN}\qquad \frac{\Gamma \vdash e : \forall t.\texttt{ST}\ t\ T}{\Gamma \vdash (\texttt{runST}\ e) : T} \quad t \notin FV(T)$$

Figure 1: Type rules

5.3 Denotational Semantics

The denotational semantics of state operations is easy to add to standard semantics for lazy functional languages. Figure 2 gives a standard semantics for a non-strict lambda calculus, except that we extend it with state transformers.

The valuation function $\mathcal{E}[\![\ ]\!]$ takes an *expression* and an *environment* and returns a *value*. We use *Env* for the domain of environments, and *Val* for the domain of values, defined as follows:

$$\begin{array}{lcl} Env & = & \prod_\tau (var_\tau \rightarrow \mathcal{D}_\tau) \\ Val & = & \bigcup_\tau \mathcal{D}_\tau \end{array}$$

The environment maps a variable of type τ to a value in the domain $\mathcal{D}_\tau$, and the domain of values is the union of all the $\mathcal{D}_\tau$, where τ ranges over monotypes.

From the point of view of the language, the type constructors `ST`, `MutVar` and `MutArr` are opaque. To give them meaning, however, the semantics must provide them with some structure.

$$\begin{array}{lcl} D_{\texttt{ST}\ s\ a} & = & State\ s \rightarrow (D_a \times State\ s) \\ D_{\texttt{MutVar}\ s\ a} & = & \mathcal{N}_\perp \\ D_{\texttt{MutArr}\ s\ a} & = & (\mathcal{N} \times \mathcal{N})_\perp \\ State\ s & = & ((\mathcal{N} \hookrightarrow Val) \times D_s)_\perp \end{array}$$

The state is a finite partial function from locations (represented by natural numbers) to values, together with a component which depends on the type of the state. The only type of values here which will concern us is `RealWorld`, the type that `s` takes in IO computations. More on this later. We also add a bottom element to represent an undefined state, and distinguish between this undefined state and states which are well-defined partial functions, but which map every thing to $\perp$. The undefined state arises naturally as the state which results from an infinite loop of state transformers, for example.

The meaning of a state transformer is a function which, given a state, produces a pair of results: a value and a new state. The least defined state transformer is the function which, given any state, returns the pair containing an undefined value and an undefined state (i.e. the bottom of the product domain).

$$\begin{array}{lcl} \mathcal{E}[\![\, Expr \,]\!] : Env \rightarrow Val & & \\ \mathcal{E}[\![\, k \,]\!]\ \rho & = & \mathcal{B}[\![\, k \,]\!] \\ \mathcal{E}[\![\, x \,]\!]\ \rho & = & \rho\ x \\ \mathcal{E}[\![\, e_1\ e_2 \,]\!] & = & (\mathcal{E}[\![\, e_1 \,]\!]\ \rho)\ (\mathcal{E}[\![\, e_2 \,]\!]\ \rho) \\ \mathcal{E}[\![\, \backslash x\texttt{->}e \,]\!]\ \rho & = & \lambda v.(\mathcal{E}[\![\, e \,]\!]\ (\rho \oplus \{x \mapsto v\}) \\ & & \\ \mathcal{E}[\![\, \texttt{runST}\ e \,]\!] & = & run\ (\mathcal{E}[\![\, e \,]\!]\ \rho) \\ \mathcal{E}[\![\, e_1\ \texttt{`thenST`}\ e_2 \,]\!] & = & bind\ (\mathcal{E}[\![\, e_1 \,]\!]\ \rho)\ (\mathcal{E}[\![\, e_2 \,]\!]\ \rho) \\ \mathcal{E}[\![\, \texttt{returnST}\ e \,]\!] & = & unit\ (\mathcal{E}[\![\, e \,]\!]\ \rho) \\ \mathcal{E}[\![\, \texttt{fixST}\ e \,]\!] & = & loop\ (\mathcal{E}[\![\, e \,]\!]\ \rho) \end{array}$$

$$\begin{array}{lcl} run\ m & = & \pi_1\ (m\ (\emptyset, \perp)) \\ (bind\ m\ k)\ \sigma & = & k\ x\ \sigma'\ \textbf{where}\ (x, \sigma') = m\ \sigma \\ (unit\ v)\ \sigma & = & (v, \sigma) \\ (loop\ f)\ \sigma & = & f\ x\ \sigma \\ & & \quad \textbf{where}\ x = fix\ (\lambda y.\pi_1\ (f\ y\ \sigma)) \end{array}$$

Figure 2: Semantics of State Combinators

References are denoted simply by natural numbers, except that it is possible to have an undefined reference also, denoted by $\perp$. The number represents a "location" in the state. Arrays are located by a pair of naturals representing the location of element 0, and the size of the array, but again it is possible to have a totally undefined array reference.

Turning to the details of Figures 2 to 5, we note a number

$$
\begin{array}{lcll}
\mathcal{E}[\![\, \texttt{newVar}\ e_1 \,]\!] & = & newVar\ v_1 & \\
\mathcal{E}[\![\, \texttt{readVar}\ e_1 \,]\!] & = & \begin{cases} \bot, & \textbf{if } v_1 = \bot \\ readVar\ v_1 & \textbf{otherwise} \end{cases} & \textbf{where } v_i = \mathcal{E}[\![\, e_i \,]\!]\ \rho \\
\mathcal{E}[\![\, \texttt{writeVar}\ e_1\ e_2 \,]\!] & = & \begin{cases} \bot, & \textbf{if } v_1 = \bot \\ writeVar\ v_1\ v_2 & \textbf{otherwise} \end{cases} &
\end{array}
$$

$$
\begin{array}{lcll}
(newVar\ v)\ \ \bot & = & (\bot,\ \bot) & \\
(newVar\ v)\ (\tau, w) & = & (p,\ (\tau[p \mapsto v], w))\ \ \textbf{where } p \notin dom(\tau) & \\
 & & & \\
(readVar\ p)\ \ \bot & = & (\bot,\ \bot) & \\
(readVar\ p)\ (\tau, w) & = & \begin{cases} (\bot,\ \bot), & \textbf{if } p \notin dom(\tau) \\ (\tau\ p,\ (\tau, w)), & \textbf{otherwise} \end{cases} & \textbf{where } v_i = \mathcal{E}[\![\, e_i \,]\!]\ \rho \\
(writeVar\ p\ v)\ \ \bot & = & (\bot,\ \bot) & \\
(writeVar\ p\ v)\ (\tau, w) & = & \begin{cases} (\bot,\ \bot), & \textbf{if } p \notin dom(\tau) \\ ((),\ \tau[p \mapsto v]), & \textbf{otherwise} \end{cases} &
\end{array}
$$

Figure 3: Semantics of variables

$$
\begin{array}{lcll}
\mathcal{E}[\![\, \texttt{newArr}\ e_1\ e_2 \,]\!] & = & \begin{cases} \bot, & \textbf{if } v_1 = \bot \\ newArr\ v_1\ v_2 & \textbf{otherwise} \end{cases} & \\
\mathcal{E}[\![\, \texttt{readArr}\ e_1\ e_2 \,]\!] & = & \begin{cases} \bot, & \textbf{if } v_1 = \bot \textbf{ or } v_2 = \bot \\ readArr\ v_1\ v_2 & \textbf{otherwise} \end{cases} & \textbf{where } v_i = \mathcal{E}[\![\, e_i \,]\!]\ \rho \\
\mathcal{E}[\![\, \texttt{writeArr}\ e_1\ e_2\ e_3 \,]\!] & = & \begin{cases} \bot, & \textbf{if } v_1 = \bot \textbf{ or } v_2 = \bot \\ writeArr\ v_1\ v_2\ v_3 & \textbf{otherwise} \end{cases} & \\
\mathcal{E}[\![\, \texttt{freezeArr}\ e_1 \,]\!] & = & \begin{cases} \bot, & \textbf{if } v_1 = \bot \\ freezeArr v_1 & \textbf{otherwise} \end{cases} &
\end{array}
$$

$$
\begin{array}{lcl}
(newArr\ n\ v)\ \ \bot & = & (\bot, \bot) \\
(newArr\ n\ v)\ (\tau, w) & = & ((p, n),\ (\tau[p \mapsto v, \ldots, (p+n-1) \mapsto v], w)) \\
 & & where\ \forall q : \{p \ldots (p+n-1)\}.q \notin dom(\tau) \\
 & & \\
(readArr\ (p, n)\ i)\ \ \bot & = & (\bot, \bot) \\
(readArr\ (p, n)\ i)\ (\tau, w) & = & \begin{cases} (\bot, \bot), & \textbf{if } i \notin \{0 \ldots (n-1)\} \textbf{ or } p + i \notin dom(\tau) \\ (\tau\ (p+i),\ (\tau, w)), & \textbf{otherwise} \end{cases} \\
 & & \\
(writeArr\ (p, n)\ i\ v)\ \ \bot & = & (\bot, \bot) \\
(writeArr\ (p, n)\ i\ v)\ (\tau, w) & = & \begin{cases} (\bot, \bot), & \textbf{if } i \notin \{0 \ldots (n-1)\} \textbf{ or } p + i \notin dom(\tau) \\ ((),\ (\tau[p + i \mapsto v], w)), & \textbf{otherwise} \end{cases} \\
 & & \\
(freezeArr\ (p, n))\ \ \bot & = & (\bot, \bot) \\
(freezeArr\ (p, n))\ (\tau, w) & = & \begin{cases} (\bot, \bot), & \textbf{if } \exists q : \{p \ldots (p+n-1)\}.q \notin dom\tau \\ (\{i \mapsto \tau(p+i)\}_{i=0}^{n},\ (\tau, w)), & \textbf{otherwise} \end{cases}
\end{array}
$$

Figure 4: Semantics of Arrays

of aspects. Some of the operations are strict. For example, `readVar`, and `writeVar` are strict in their references, but none of the operations are strict in the values stored. Again, `newVar`, `readVar`, and `writeVar` are strict in the state, but `thenST` and `returnST` are not. We shall return to this point in a moment.

Array references are treated similarly, except that they come in two parts: the base reference and an offset (the array index). We assume indexing ranges from 0 to the size of the array minus 1. Again, `readArr` and `writeArr` are strict in both the array reference and in the index. They also return undefined if the index is out of bounds. Finally, they and `newArr` are strict in the state. `newArr` is also strict in the array size.

The semantics makes the nature of arrays crystal clear. They are composed of many separate locations in the state, each independently updatable, each update costing no more than a variable update would cost.

$$
\begin{array}{l}
\mathcal{E}[\![\, \texttt{ccall}\ fn\ e_1 \ldots e_n \,]\!] \\
\quad = \left\{ \begin{array}{ll} \bot, & \textbf{if } \exists i. v_i = \bot \\ doIO\ fn\ [v_1, \ldots, v_n] & \textbf{otherwise} \end{array} \right. \\
\quad \textbf{where } v_i = \mathcal{E}[\![\, e_i \,]\!]\ \rho
\end{array}
$$

$$
\begin{array}{l}
doIO : Name \rightarrow SeqVal \rightarrow \\
\quad State\ RealWorld \rightarrow (Val \times State\ RealWord) \\
doIO\ fn\ args\ \bot \quad = \quad (\bot,\ \bot) \\
doIO\ fn\ args\ (\tau, rw) \quad = \quad (v,\ (\tau, rw')) \\
\quad \textbf{where } (v, rw') = \mathcal{IO}\ fn\ args\ rw
\end{array}
$$

$$
\begin{array}{l}
\mathcal{IO} : Name \rightarrow Seq\ Val \rightarrow \\
\quad RealWorld \rightarrow (Val \times RealWorld)
\end{array}
$$

Figure 5: Semantics of IO operations

The semantics of IO operations is given with respect to an unspecified function $\mathcal{IO}$ which is a state transformer on the `RealWorld`.

The primitive operations, such as `newVar`, `readVar`, and so on, are necessarily strict in the state. After all, they each depend on the value of the state. In contrast, the semantics do not make `thenST` or `runST` strict in the state, since they do not need its value.

What difference might this make? Consider the interpreter for a language with IO operations and state given in Figure 6. The values of variables are stored in a mutable array, and a variable is used to store the input (a lazy list). The result of obeying the commands is a list of the output values. The output should appear as it is generated: whenever a `Write` is obeyed, the `returnST` should be able to make the first element of its output list available to the outside world *before* obeying the rest of the commands.

5.4 Safety

How can we be sure that the above type system ensures that each state thread is independent of all others? A full proof lies well outside the scope of this paper, but we will provide an outline of the ideas, together with a (slightly informal) statement of the main result.

Our ultimate intention is to implement all the distinct state threads using the single, global, machine state, but to do so without affecting the values returned by state computations. That is, we want to be able to map each thread on to the global state, and to ensure that the individual state threads cannot communicate with each other except through purely functional values. Similarly, we want to guarantee that we can arbitrarily interleave the evaluation of separate state threads, still without affecting the result of the computation because, in practice, the amount of computation which takes place in each thread will depend on the demand for its final value. We certainly do not want the value of the result to depend on the pattern of demand.

Consider then a single state thread. In the semantics it is applied to the empty state, and this state is extended by applications of `newVar` (and `newArr` of course). There are no other active locations, and the state undergoes no transformations other than those specified by these reads and writes.

In contrast, if we were to imagine using a single global state for *all* the threads, then many more locations would be active and, because of possible interleavings of operations from other threads, the underlying state might change while passing from one primitive operation to the next. The big question is: how can we guarantee that these changes do not affect the result of the state thread?

We view the state from the perspective of a single thread, `runST m`, and model the changes made by other threads as non-deterministic changes to the state. So, first, we have to map the state of this particular thread into a global state containing material from many separate threads. We model this mapping by an injection (one-to-one mapping) $\phi : \mathcal{N} \rightarrow \mathcal{N}$ which maps a reference from the local state of `m` to a reference in a global state. The range of ϕ is that part of the global address space which supports the execution of `m`.

We will say that a global state τ_1 models the local state of the semantics, τ_m, if $\tau_m = \tau_1 \circ \phi$. That is, if τ_m is defined at some location v to have a value x, then τ_1 is defined at location $\phi\ x$ also having value x. In addition, however, τ_1 may be defined at lots of other locations outside the range of ϕ.

We can also relate two global states using ϕ: two global states are related by ϕ if they both model the same local state. That is, $\widehat{\phi} : \tau_0 \leftrightarrow \tau_1$ is a relation defined by $\tau_0 \circ \phi = \tau_1 \circ \phi$. So, two global states are related by $\widehat{\phi}$ if they agree in all the positions in the range of ϕ, but they can have any old junk in all other locations.

Now, when we actually evaluate `runST m`, we will allocate locations only in the range of ϕ. Of course, ϕ only exists as a mathematical abstraction. The allocation mechanism will merely choose a free location: ϕ acts as a description of how the allocation mechanism will have behaved.

Each separate thread will allocate references in non-overlapping areas (otherwise the locations would not be available). So from the perspective of a single thread, the possible changes to the references belonging to other threads can be described by allowing the state to be *any* state which happens to model the true local state by ϕ.

Now we can almost state the main correctness theorem. To do so, we write m for the meaning of a state transformer `m` as given by the semantics, and m_ϕ as the meaning of `m` assuming states are referenced via the coding function ϕ. That is, the same semantics, except that *bind* and *newVar* are replaced with $bind_\phi$ and $newVar_\phi$

$$
\begin{aligned}
&(bind_\phi\ m\ k)\ \sigma = k\ x\ \sigma'' \\
&\quad \textbf{where}\quad (x, \sigma') = m\ \sigma \\
&\qquad\qquad\quad\ \sigma'\ \widehat{\phi}\ \sigma''
\end{aligned}
$$

$$
\begin{aligned}
&(newVar_\phi\ v)\ (\tau, w) = (p,\ (\tau[p \mapsto v], w)) \\
&\quad \textbf{where}\ p \notin dom(\tau) \wedge p \in ran(\phi)
\end{aligned}
$$

The only differences are that *newVar* is only allowed to allocate locations in the range of ϕ, and that *bind* can alter its intermediate state so long as the status of the locations in the range of ϕ are preserved.

Theorem. If $m : \forall s. ST\ s\ T$ (where $s \notin FV(T)$) then for any injection $\phi : \mathcal{N} \to \mathcal{N}$, and any $\tau : \mathcal{N} \hookrightarrow Val$ such that $\tau \circ \phi = \emptyset$, we have $\pi_1\ (m\ (\emptyset, w)) = \pi_1\ (m_\phi\ (\tau, w))$, for all real worlds w.

The proof uses parametric polymorphism in the style of Mitchell & Meyer [1985].

The theorem says that we can choose any initial state, so long as nothing is defined in the range of ϕ, and the final result is the same as when the semantics explicitly used purely local state. Furthermore, the state can change in each use of `thenST` (modelled by *bind*) so long as nothing in the range of ϕ is touched and, again, the final result is unchanged.

The corollary to this is that no state thread can read a reference allocated by another thread (otherwise the result of `runST m` *would* depend on the choice of initial state, or on how the other parts of the state changed within an application of *bind*). Similarly, no state thread can write to a reference belonging to another thread because the result of *writeVar* depends on whether the location is allocated or not, but again, the result of `runST m` is independent of such matters.

In conclusion, therefore, each thread is independent of other threads, even when implemented in a single global store.

6 Implementation

The whole point of expressing stateful computations in the framework that we have described is that operations which modify the state can update the state *in place*. The implementation is therefore crucial to the whole enterprise, rather than being a peripheral issue.

We have in mind the following implementation framework:

- The *state* of each encapsulated state thread is represented by a collection of objects in heap-allocated storage.
- A *reference* is represented by the address of an object in heap-allocated store.
- A *read operation* returns the current contents of the object whose reference is given.
- A *write operation* overwrites the contents of the specified object or, in the case of mutable arrays, part of the contents.
- The I/O thread is a little different because, as discussed in Section 5.3, its state also includes the actual state of the real world. I/O operations are carried out directly on the real world (updating it in place, as it were).

As the previous section outlined, the correctness of this implementation relies totally on the type system. Such a reliance is quite familiar: for example, the implementation of addition makes no attempt to check that its arguments are indeed integers, because the type system ensures it. In the same way, the implementation of state transformers makes no attempt to ensure, for example, that references are only used in the same state thread in which they were created; the type system ensures that this is so.

6.1 Update in place

The most critical correctness issue concerns the update-in-place behaviour of write operations. Why is update-in-place safe? It is safe because all the combinators (**thenST**, **returnST**, **fixST**) use the state only in a single-threaded manner (Schmidt [1985]); that is, they each use the incoming state exactly once, and none duplicates it (Figure 2). Furthermore, all the primitive operations on the state are strict in it. A write operation can modify the state in place, because (a) it has the only copy of the incoming state, and (b) since it is strict in the incoming state, there can be no as-yet-unevaluated read operations pending on that state.

Can the programmer somehow duplicate the state? No: since the `ST` type is opaque, the only way the programmer can manipulate the state is *via* the combinators **thenST**, **returnST** and **fixST**. On the other hand, the programmer certainly does have access to named references into the state. However, it is perfectly OK for these to be duplicated, stored in data structures and so on. Variables are *immutable*; it is only the state to which they refer that is altered by a write operation.

We find these arguments convincing, but they are certainly not formal. A formal proof would necessarily involve some operational semantics, and a proof that no evaluation order could change the behaviour of the program. We have not yet undertaken such a proof.

6.2 Efficiency considerations

It would be possible to implement state transformers by providing the combinators (`thenST`, `returnST`, etc) and primitive operations (`readVar`, `writeVar` etc) as library functions. But this would impose a very heavy overhead on each operation and (worse still) on composition. For example, a use of `thenST` would entail the construction of two function-valued arguments, followed by a procedure call to `thenST`. This compares very poorly with simple juxtaposition of code, which is how sequential composition is implemented in conventional languages!

A better way would be to treat state-transformer operations specially in the code generator. But that risks complicating an already complex part of the compiler. Instead we implement state transformers in a way which is both direct and efficient: we simply give Haskell definitions for the combinators.

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

```
returnST x s = (x,s)
thenST m k s = k x s' where (x,s') = m s

fixST k s = (r,s') where (r,s') = k r s
runST m = r where (r,s) = m currentState
```

Rather than provide `ST` as a built-in type, opaque to the compiler, we give its representation with an explicit Haskell type definition. (The representation of `ST` is not, of course, exposed to the programmer, lest he or she write functions which duplicate or discard the state.) It is then easy to give Haskell definitions for the combinators, which mirror precisely the semantics given for them in Figure 2[4].

The implementation of **runST** is intriguing. Since its argument, `m`, works regardless of what state is passed to it, we simply pass a value representing the current state of the heap. As we will see shortly (Section 6.2.2), this value is never actually looked at, so a constant value will do.

The code generator must, of course, remain responsible for producing the appropriate code for each primitive operation, such as **readVar**, **ccall**, and so on. In our implementation we actually provide a Haskell "wrapper" for each primitive which makes explicit the evaluation of their arguments, using so-called "unboxed values". Both the motivation for and the implementation of our approach to unboxed values is detailed in Peyton Jones & Launchbury [1991], and we do not rehearse it here.

6.2.1 Transformation

The beauty of this approach is that all the combinators can then be in-lined at their call sites, thus largely removing the "plumbing" costs. For example, the expression

```
m1 `thenST` \v1 ->
m2 `thenST` \v2 ->
returnST e
```

becomes, after in-lining `thenST` and `returnST`,

```
\s -> let (v1,s1) = m1 s
          (v2,s2) = m2 s1
      in (e,s3)
```

Furthermore, the resulting code is now exposed to the full range of analyses and program transformations implemented by the compiler. For example, if the compiler can spot that the above code will be used in a context which is strict in either component of the result tuple, it will be transformed to

```
\s -> case m1 s of
        (v1,s2) -> case m2 s1 of
                     (v2,s2) -> (e,s2)
```

In the `let` version, heap-allocated thunks are created for

[4] Indeed, we have to admit that the implementation came first!

`m1 s` and `m2 s1`; the `case` version avoids this cost. These sorts of optimisations could not be performed if the `ST` type and its combinators were opaque to the compiler.

6.2.2 Passing the state around

The implementation of the `ST` type, given above, passes around an explicit state. Yet, we said earlier that state-manipulating operations are implemented by performing side effects on the common, global heap. What, then, is the role of the explicit state values which are passed around by the above code? It plays two important roles.

Firstly, the compiler "shakes the code around" quite considerably: is it possible that it might somehow end up changing the order in which the primitive operations are performed? No, it is not. The input state of each primitive operation is produced by the preceding operation, so the ordering between them is maintained by simple data dependencies of the explicit state, which are certainly preserved by every correct program transformation.

Secondly, the explicit state allows us to express to the compiler the strictness of the primitive operations in the state. The `State` type is defined like this:

```
data State s = MkState (State# s)
```

That is, a state is represented by a single-constructor algebraic data type, whose only contents is a value of type `State# s`, the (finally!) primitive type of states. The lifting implied by the `MkState` constructor corresponds exactly to the lifting in the semantics. Using this definition of `State` we can now define `newVar`, for example, like this:

```
newVar init (MkState s#)
  = case newVar# init s# of
       (v,t#) -> (v, MkState t#)
```

This definition makes absolutely explicit the evaluation of the strictness of `newVar` in its state argument, finally calling the truly primitive `newVar#` to perform the allocation.

We think of a primitive state — that is, a value of type `State# s`, for some type `s` — as a "token" which stands for the state of the heap and (in the case of the I/O thread) the real world. The implementation never actually inspects a primitive state value, but it is faithfully passed to, and returned from every primitive state-transformer operation. By the time the program reaches the code generator, the role of these state values is over, and the code generator arranges to generate no code at all to move around values of type `State#` (assuming an underlying RAM architecture of course).

6.2.3 Arrays

The implementation of arrays is straightforward. The only complication lies with `freezeArray`, which takes a mutable array and returns a frozen, immutable copy. Often, though, we want to construct an array incrementally, and then freeze it, performing no further mutation on the mutable array. In this case it seems rather a waste to copy the entire array, only to discard the mutable version immediately thereafter.

The right solution is to do a good enough job in the compiler to spot this special case. What we actually do at the moment is to provide a highly dangerous operation `dangerousFreezeArray`, whose type is the same as `freezeArray`, but which works without copying the mutable array. Frankly this is a hack, but since we only expect to use it in one or two critical pieces of the standard library, we couldn't work up enough steam to do the job properly just to handle these few occasions. We do not provide general access to `dangerousFreezeArray`.

6.2.4 More efficient I/O

The I/O state transformer is a little special, because of the following observation: *the final state of the I/O thread will certainly be demanded.* Why? Because the whole point in running the program in the first place is to cause some side effect on the real world!

We can exploit this property to gain a little extra efficiency. Since the final state of the I/O thread will be demanded, so will every intermediate thread. So we can safely use a strict, and hence more efficient, version of `thenST`:

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

By using `case` instead of the `let` which appears in `thenST`, we avoid the construction of a heap-allocated thunk for `m s`.

7 Other useful combinators

We have found it useful to expand the range of combinators and primitives beyond the minimal set presented so far. This section presents the ones we have found most useful.

7.1 Equality

The references we have correspond very closely to "pointers to variables". One useful additional operation on references is to determine whether two references are aliases for the same variable (so *writes* to the one will affect *reads* from the other). It turns out to be quite straightforward to add an additional constant,

```
eqMutVar :: MutVar s a -> MutVar s a -> Bool
eqMutArr :: Ix i =>
        MutArr s i a -> MutArr s i a -> Bool
```

Notice that the result does *not* depend on the state—it is simply a boolean. Notice also that we only provide a test on references which exist in the same state thread. References from different state threads cannot be aliases for one another.

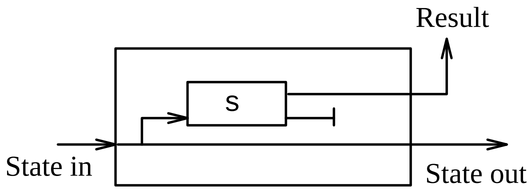
7.2 Interleaved and parallel operations

The state-transformer composition combinator defined so far, `thenST`, is strictly sequential: the state is passed from the first state transformer on to the second. But sometimes that is not what is wanted. Consider, for example, the operation of reading a file. We may not want to specify the precise relative ordering of the individual character-by-character reads from the file and other I/O operations. Rather, we may want the file to be read lazily, as its contents is demanded.

We can provide this ability with a new combinator, `interleaveST`:

```
interleaveST :: ST s a -> ST s a
```

Unlike
every other state transformer so far, `interleaveST` actually duplicates the state! The "plumbing diagram" for `(interleaveST s)` is like this:



More precisely, `interleaveST` splits the state into two parts, which should be disjoint. In the lazy-file-read example, the state of the file is passed into one branch, and the rest of the state of the world is passed into the other. Since these states are disjoint, an arbitrary interleaving of operations in each branch of the fork is legitimate.

To make all this concrete, here is an implementation of lazy file read:

```
readFile :: String -> IO [Char]
readFile filename
  = openFile filename  `thenST` \f ->
    readCts f

readCts :: FileDescriptor -> IO [Char]
readCts f = interleaveST
  (readCh f  `thenST` \c ->
   if c == eofChar
   then returnST []
   else readCts f  `thenST` \cs ->
        returnST (c:cs))
```

A parallel version of `interleaveST`, which starts up a concurrent task to perform the forked I/O thread, seems as though it would be useful in building responsive graphical user interfaces. The idea is that `forkIO` would be used to create a new widget, or window, which would be capable of independent I/O through its part of the screen.

The only unsatisfactory feature of all this is that we see absolutely no way to guarantee that the side effects performed in the two branches of the fork are indeed independent. That has to be left as a proof obligation for the programmer; the only consolation is that at least the location of these proof obligations is explicit. We fear that there may be no absolutely secure system which is also expressive enough to describe the programs which real programmers want to write.

8 Related work

Several other languages from the functional stable provide some kind of state.

For example, Standard ML provides *reference types*, which may be updated (Paulson [1991]). The resulting system has serious shortcomings, though. The meaning of programs which use references depends on a complete specification of the order of evaluation of the program. Since SML is strict this is an acceptable price to pay, but it would become unworkable in a non-strict language where the exact order of evaluation is hard to figure out. What is worse, however, is that referential transparency is lost. Because an arbitrary function may rely on state accesses, its result need not depend purely on the values of its arguments. This has additional implications for polymorphism, leading to a weakened form in order to maintain type safety (Tofte [1990]). We have none of these problems here.

The dataflow language Id provides I-structures and M-structures as mutable datatypes (Nikhil [1988]). Within a stateful program referential transparency is lost. For I-structures, the result is independent of evaluation order, provided that all sub-expressions are eventually evaluated (in case they side-effect an I-structure). For M-structures, the result of a program can depend on evaluation order. Compared with I-structures and M-structures, our approach permits lazy evaluation (where values are evaluated on demand, and may never be evaluated if they are not required), and supports a much stronger notion of encapsulation. The big advantage of I-structures and M-structures is that they are better suited to parallel programming than is our method.

The Clean language takes a different approach (Barendsen & Smetsers [1993]). The Clean type system supports a form of linear types, called "unique types". A value whose type is unique can safely be updated in place, because the type system ensures that the updating operation has the sole reference to the value. The contrast with our work is interesting. We separate references from the state to which they refer, and do not permit explicit manipulation of the state. Clean identifies the two, and in consequence requires state to be manipulated explicitly. We allow references to be duplicated, stored in data structures and so on, while Clean does not. Clean requires a new type system to be explained to the programmer, while our system does not. On the other hand, the separation between references and state is sometimes tiresome. For example, while both systems can express the idea of a mutable list, Clean does so more neatly because there is less explicit de-referencing. The tradeoff between implicit and explicit state in purely-functional languages is far from clear.

There are significant similarities with Gifford and Lucassen's *effect system* which uses types to record side effects performed by a program (Gifford & Lucassen [1986]). However, the effects system is designed to delimit the effect of side effects which may occur as a result of evaluation. Thus the semantic setting is still one which relies on a predictable order of evaluation.

Our work also has strong similarities with Odersky, Rabin and Hudak's λ_{var} (Odersky, Rabin & Hudak [1993]), which itself was influenced by the Imperative Lambda Calculus (ILC) of Swarup, Reddy & Ireland [1991]. ILC imposed a rigid stratification of applicative, state reading, and imperative operations. The type of `runST` makes this stratification unnecessary: state operations can be encapsulated and appear purely functional. This was also true of λ_{var} but there it was achieved only through run-time checking which, as a direct consequence, precludes the style of lazy state given here.

In two earlier papers, we describe an approach to these issues based on *monads*, in the context of non-strict, purely-functional languages. The first, Peyton Jones & Wadler [1993], focusses mainly on input/output, while the second, Launchbury [1993], deals with stateful computation within a program. The approach taken by these papers has two major shortcomings:

- State and input/output existed in separate frameworks. The same general approach can handle both but, for example, different combinators were required to compose stateful computations from those required for I/O-performing computation.
- State could only safely be handled if it was *anonymous*. Consequently, it was difficult to write programs which manipulate more than one piece of state at once. Hence, programs became rather "brittle": an apparently innocuous change (adding an extra updatable array) became difficult or impossible.
- Separate state threads required expensive run-time checks to keep them apart. Without this, there was the possibility that a reference might be created in one stateful thread, and used asynchronously in another, which would destroy the Church-Rosser property.

Acknowledgements

The idea of adding an extra type variable to state threads arose in discussion with John Hughes, and was presented briefly at the 1993 Copenhagen workshop on State in Programming Languages, though at that time we suggested using an existential quantification in the type of `runST`. In addition, all these ideas have benefited from discussions amongst the Functional Programming Group at Glasgow.

References

E Barendsen & JEW Smetsers [Dec 1993], "Conventional and uniqueness typing in graph rewrite systems," in *Proc 13th Conference on the Foundations of Software Technology and Theoretical Computer Science*, Springer Verlag LNCS.

DK Gifford & JM Lucassen [Aug 1986], "Integrating functional and imperative programming," in *ACM Conference on Lisp and Functional Programming, MIT*, ACM, 28–38.

J Launchbury [June 1993], "Lazy imperative programming," in *Proc ACM Sigplan Workshop on State in Programming Languages, Copenhagen (available as YALEU/DCS/RR-968, Yale University)*, pp46–56.

NJ McCracken [June 1984], "The typechecking of programs with implicit type structure," in *Semantics of data types*, Springer Verlag LNCS 173, 301–315.

JC Mitchell & AR Meyer [1985], "Second-order logical relations," in *Logics of Programs*, R Parikh, ed., Springer Verlag LNCS 193.

Rishiyur Nikhil [March 1988], "Id Reference Manual," Lab for Computer Sci, MIT.

M Odersky, D Rabin & P Hudak [Jan 1993], "Call by name, assignment, and the lambda calculus," in *20th ACM Symposium on Principles of Programming Languages, Charleston*, ACM, 43–56.

LC Paulson [1991], *ML for the working programmer*, Cambridge University Press.

SL Peyton Jones & J Launchbury [Sept 1991], "Unboxed values as first class citizens," in *Functional Programming Languages and Computer Architecture, Boston*, Hughes, ed., LNCS 523, Springer Verlag, 636–666.

SL Peyton Jones & PL Wadler [Jan 1993], "Imperative functional programming," in *20th ACM Symposium on Principles of Programming Languages, Charleston*, ACM, 71–84.

CG Ponder, PC McGeer & A P-C Ng [June 1988], "Are applicative languages inefficient?," *SIGPLAN Notices* 23, 135–139.

DA Schmidt [Apr 1985], "Detecting global variables in denotational specifications," *TOPLAS* 7, 299–310.

V Swarup, US Reddy & E Ireland [Sept 1991], "Assignments for applicative languages," in *Functional Programming Languages and Computer Architecture, Boston*, Hughes, ed., LNCS 523, Springer Verlag, 192–214.

M Tofte [Nov 1990], "Type inference for polymorphic references," *Information and Computation* 89.

Appendix

Figure 6 gives a larger example of array references in use. It defines an interpreter for a simple imperative language, whose input is the program together with a list of input values, and whose output is the list of values written by the program. The interpreter naturally involves a value representing the state of the store. The idea is, of course, that the store should be implemented as an in-place-updated array, and that is precisely what is achieved[5].

This example has long been a classic test case for systems which infer single-threadedness (Schmidt [1985]). The only unsatisfactory feature of the solution is that `eval` has to be written as a fully-fledged state transformer, while one might perhaps like to take advantage of its "read-only" nature.

[5] This program also exhibits an awkward shortcoming of Haskell's type signatures, which it shares with every other widely-used functional language we know. The type signatures for `obey` and `eval` are given in comments only, because Haskell understands them as implicitly universally quantified over `s`. But of course they are monomorphic in `s`! Alas.

```
data Com = Assign Var Exp | Read Var | Write Exp | While Exp [Com]
type Var = Char
data Exp = ....

interpret :: [Com] -> [Int] -> [Int]
interpret cs input = runST (newArr ('A','Z') 0  `thenST` \store ->
                            newVar input        `thenST` \inp->
                            command cs store inp)

command :: [Com] -> MutArray s Int -> MutVar s [Int] -> ST s [Int]
command cs store inp = obey cs
  where
    -- obey :: [Com] -> ST s [Int]
    obey [] = returnST []
    obey (Assign v e:cs) = eval e               `thenST` \val->
                           writeArr store v val `thenST_`
                           obey cs
    obey (Read v:cs)     = readVar inp  `thenST` \(x:xs) ->
                           writeArr store v x   `thenST_`
                           writeVar inp xs      `thenST_`
                           obey cs
    obey (Write e:cs)    = eval e               `thenST` \out->
                           obey cs              `thenST` \outs->
                           returnST (out:outs)
    obey (While e bs:cs) = eval e               `thenST` \val->
                           if val==0 then
                             obey cs
                           else
                             obey (bs ++ While e bs : cs) inp

    -- eval :: Exp -> ST s Int
    eval e = ....
```

Figure 6: An interpreter with lazy stream output