, , 1–51 ()
© Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

State in Haskell

JOHN LAUNCHBURY jl@cse.ogi.edu
Oregon Graduate Institute, PO Box 91000, Portland, OR 97291-1000

SIMON L PEYTON JONES simonpj@dcs.glasgow.ac.uk
University of Glasgow, G12 8QQ, Scotland

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. The same framework is also used to handle input/output operations (state changes on the external world) and calls to C.

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]). Examples of such algorithms include:

- Algorithms based on the use of incrementally-modified hash tables, where lookups are interleaved with the insertion of new items.

- The union/find algorithm, which relies for its efficiency on the set representations being simplified each time the structure is examined.

- Many graph algorithms, which 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 change to 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 manipulation.

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[1]. 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 and, of course, its behaviour may depend on that state.

  But in addition, we guarantee that the state is used in a single-threaded way. Consequently, 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.

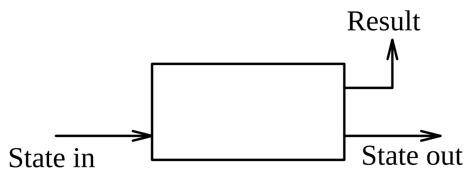
- Stateful computations can even be performed *lazily* without losing safety. For example, suppose that stateful depth-first search of a graph returns a list of vertices in depth-first order. If the consumer of this list only evaluates the first few elements of the list, then only enough of the stateful 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 *state transformer* 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 pure function, taking a state as its argument, and delivering a state and a value as its result. We depict a state transformer like this:

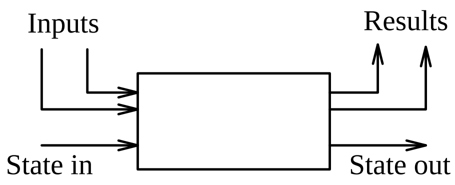


From a semantic point of view, this is a purely-functional account of state. For example, being a pure function, a state transformer is a first-class value: it can be passed to a function, returned as a result, stored in a data structure, duplicated freely, and so on. (Of course, it is our intention that the new state will actually be constructed by modifying the old one in place, a matter to which we return in Section 9.) From now on, we take the term "state transformer" to be synonymous with "stateful computation": the computation is seen as transforming one state into another.

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)
```

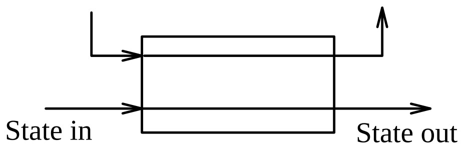
Its picture might look like this:



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. Mutable variables

Next, we need to provide some primitive state transformers which operate on the state. The simplest thing to provide is the ability to allocate, read, and write mutable variables:

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

The type `MutVar s a` is the type of *references* allocated from a state indexed by `s`, and containing a value of type `a`. 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 state contains a finite mapping of references to values.

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, a decision whose importance will become apparent in Section 2.5. (We use the name `MutVar` for the type of references, rather than `Ref`, specifically to avoid confusion with SML.)

Returning to the primitives, 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 state transformer is applied to a state, it allocates a fresh reference — that is, one currently not used in the state — augments the state to map the new reference to the supplied value, and returns the reference along with the modified state.

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.

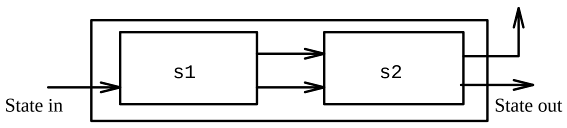
The function `writeVar` transforms the state so that it maps the given reference to a new value. Notice that *the reference itself does not change*; it is the *state* which is modified. `writeVar` delivers a result of the unit type `()`, a type which only has one value (apart from bottom), also written `()`. 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[2]:



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.

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)))
```

When `swap v w` is executed in a state thread (that is, when applied to 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.

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 is just as you would expect: the second argument of a `thenST` is meant to be a function. Furthermore, the parentheses enclosing the lambda abstractions can be omitted, and we will do so from now on. They would only be required if we wanted the lambda abstraction to extend less far than the end of the expression. Lastly, 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.

2.4. Other useful combinators

To avoid the frequent appearance of lambda abstractions with wild-card patterns, we provide a special form of `thenST`, called `thenST_`, with the following definition:

```
thenST_ :: ST s () -> ST s b -> ST s b
thenST_ st1 st2 = st1 `thenST` \ _ ->
                  st2
```

Unlike `thenST`, the second argument of `thenST_` is not a function, so the lambda isn't required. Using `thenST_`, and omitting parentheses, we can rewrite `swap` more tidily 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
```

Four other useful combinators, definable in terms of `thenST` and `returnST`, are `listST`, `listST_`, `mapST`, and `mapST_`:

```
listST :: [ST s a] -> ST s [a]
listST sts = foldr consST nilST sts
      where
        nilST = returnST []
        consST m ms = m       `thenST` \ r ->
                      ms      `thenST` \ rs ->
                      returnST (r:rs)

listST_ : [ST s a] -> ST s ()
listST_ sts = foldr thenST_ (returnST ()) sts

mapST :: (a -> ST s b) -> [a] -> ST s [b]
mapST f xs = listST (map f xs)
```

```
mapST_ :: (a -> ST s ()) -> [a] -> ST s ()
mapST_ f xs = listST_ (map f xs)
```

The first, `listST` takes a list of state transformers, each indexed by the same state type and returning a value of the same type. It glues them together into a single state transformer, which composes its components together in sequence, and collects their results into a list. The other three are useful variants. `mapST`, for example, is rather like the normal `map` function — it applies a function repeatedly to the elements of a list, but then it also feeds the state through the resulting state transformers.

2.5. 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. Here is a tiny example of `runST` in action:

```
three :: Int
three = runST (newVar 0       `thenST` \ v ->
               writeVar v 3   `thenST_`
               readVar v)
```

This definition of the value `three`, runs a state thread which allocates a new variable, writes `3` into it, reads the variable, and returns the value thus read. (Sections 5 and 7 gives some more convincing uses of `runST`.)

2.5.1. A flaw

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 runtime check. The obvious idea is to allocate a unique *state-thread identifier* with each call to `runST`, which is carried along in the state. Every reference would include the identifier of the thread in which it was created, and whenever a reference was used for reading or writing, a runtime check is made to ensure that the identifier in the reference matches that in the state.

This sounds easy enough, albeit perhaps inefficient. The trouble is that it does not work! Consider the following (recursive) definition

```
foo = runST (newVar 7         `thenST` \ v ->
             writeVar foo 3   `thenST_`
             returnST v)
```

Does this give a runtime error? No, the write is in the same thread as the allocate. However, the following pair of mutually-recursive definitions ought to behave identically:

```
foo1 = runST (newVar 7        `thenST` \ v ->
              writeVar foo2 3 `thenST_`
              returnST v)
foo2 = runST (newVar 7        `thenST` \ v ->
              writeVar foo1 3 `thenST_`
              returnST v)
```

All that we have done is to unfold the definition of `foo` once, which certainly should not change the semantics of the program. But alas, each write sees a variable from the "other" thread, so the runtime check will fail. A perfectly respectable program transformation has changed the behaviour of the runtime check, which is unacceptable.

2.5.2. The solution: parametricity

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)
```

A `runST` with this type could legally be applied to any state transformer, regardless of the types to which `s` and `a` are instantiated. However, we can be more precise:

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 the type of `(newVar True)`. Indeed there is no instance of `runST` which can be applied to `(newVar True)`.

Just to demonstrate that the type of `runST` does nevertheless permit one thread to manipulate references belonging to another thread, here is a perfectly legitimate (albeit artificial) example:

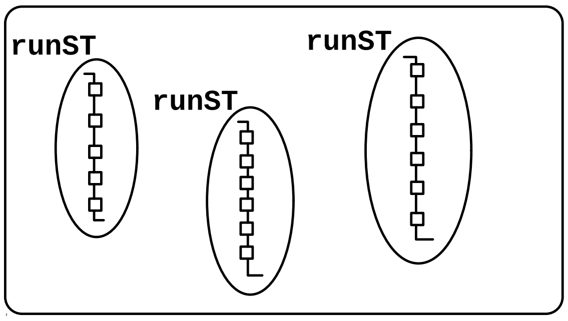


Figure 1. An informal picture of a program with state threads

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

where `v` is a reference from some (other) 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 Sections 6 and 9.2, where we show that `runST`'s type can be accommodated with only a minor enhancement to the type checker.

2.6. Summary

We have described a small collection of:

- Primitive state transformers, namely `returnST`, `newVar`, `readVar`, and `writeVar`;
- "Plumbing" combinators, which compose state transformers together, namely `thenST` and its derivatives, `thenST_`, `listST`, `mapST`, and so on.
- An encapsulator, `runST`, which runs a state transformer on the empty state, discards the resulting state, and returns the result delivered by the state transformer.

Figure 1 gives an informal picture of a program with a number of calls to `runST`. Each call to `runST` gives rise to an independent thread, depicted as a large oval.

The plumbing combinators ensure that the state is single-threaded, so that the net effect is to plumb the state through a chain of primitive operations (`newVar`, `readVar` and `writeVar`), each of which is depicted as a small square box. The relative interleaving of the primitive operations in each thread is undefined but, since the threads share no state, different interleavings cannot give rise to different results.

Notice that it is only the state that is single-threaded. Both references and state transformers, in contrast, are first-class values which can be duplicated, discarded, stored in data structures, passed to functions, returned as results, and so on. A reference can only be used, however, by bringing it back together with "its" state in a `readVar` or `writeVar` operation.

The crucial idea is that of "indexing" state transformers, states, and mutable variables, with a type. It is usual for a value of type `Tree t`, say, to contain sub-components of type `t`. That is not the case for mutable variables and state transformers! In the type `ST s a`, the "`s`" is used to label the state transformer, and force compatibility between state transformers which are composed together. Similarly, the "`s`" in `MutVar s a` simply ensures that a reference can only be dereferenced in "its" state thread. It is for this reason that we speak of a state "being indexed by" a type, rather than "having" that type.

The type constructor `ST` together with the functions `returnST` and `thenST` form a so-called *monad* (Moggi [1989]), and have a nice algebra. For a detailed discussion of monads see Wadler [1992a].

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 operations to allocate, read and write elements of arrays. They have the following types[3]:

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

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 of type `MutArr s i v`, 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 three operations can be implemented using the mutable variables already introduced, by representing an array reference as an (immutable) array of references, thus:

```
type MutArr s i v = Array i (MutVar s v)
```

This definition makes use of the standard Haskell type `Array i v`, the type of arrays indexed by values of type `i`, with elements of type `v`. Haskell arrays are constructed by the function `array`, and indexed with the infix operator `(!)`, whose types are as follows:

```
array :: Ix i => (i,i) -> [(i,v)]  -> Array i v
(!)   :: Ix i => Array i v -> i -> v
```

The array-construction function `array` takes the bounds of the array and list of index-value pairs, and returns an array constructed by filling in the array in the way specified by the list of index-value pairs[4].

The implementation of `newArr`, `readArr`, and `writeArr` is then straightforward:

```
newArr bds init = mapST newVar indices    `thenST` \ vs ->
                  returnST (array bds (indices `zip` vs))
         where
           -- indices :: [(i,v)]
           indices = range bds

readArr  arr i   = readVar  (arr!i)
writeArr arr i v = writeVar (arr!i) v
```

The only interesting definition is that for `newArr`, which uses `mapST` (Section 2.4) to apply `newVar` to each index. The standard function `range` is applied to the bounds of the array — that is, a pair of values in class `Ix` — to deliver a list of all the indices of the array[5].

In practice, we do not implement mutable arrays in this way for two reasons:

- Implementing arrays in terms of variables is rather inefficient. A single mutable array of 100 elements would require an immutable array of 100 elements, plus 100 separately-allocated mutable variables. Each read or write would require two memory references, one to the array, and a second to the variable. It would obviously be more sensible to allocate one mutable array of 100 elements, and have `writeArr` mutate the elements directly.

- Implementing arrays in terms of variables depends on the existence of standard Haskell arrays. How are the latter to be implemented? Presumably, in a sequential system, the implementation of `array` will involve allocating a suitable block of memory, and running down the list of index-value pairs, filling in the specified elements of the array as we go. Looked at like this, we need mutable arrays to implement immutable arrays! (We return to this topic in Section 7.)

Because of these considerations we actually provide `newArray`, `readArr` and `writeArr` as primitives, and define `newVar`, `readVar`, and `writeVar` in terms of them, by representing variables as arrays of size one.

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, namely `RealWorld`, 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 = mapST_ putChar cs
```

4.1. System calls

It would be possible to provide `putChar` and `getChar` as primitives — that is, functions which are not definable in Haskell. The difficulty is that there are potentially a very large collection of such "primitive" I/O operations, and it is very likely that programmers will want to add new ones of their own. To meet this concern, we provide just one primitive I/O operation, called `ccall`, which allows the Haskell programmer to call any C procedure. Using `ccall` we can define all the other I/O operations; for example, `putChar` is defined like this:

```
putChar :: Char -> IO ()
putChar c = ccall putchar c
```

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 result type of `putChar`. Similarly, `getChar` is implemented like this:

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

We implement `ccall` 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. (An ordinary function, possessing only one type, would take a fixed number of arguments of fixed type.) The restrictions placed on its use are:

- All the arguments, and the result, must be types which C understands: `Int`, `Float`, `Double`, `Bool`, `String`, or `Array`. There is no automatic conversion of more complex structured types, such as lists (apart from lists of characters, which are treated specially as strings) or trees. Furthermore, the types involved must be statically deducible by the compiler from the surrounding context; that is, `ccall` cannot be used in a polymorphic context. The programmer can easily supply any missing type information with a type signature.
- The first "argument" of `ccall`, which is the name of the C function to be called, must appear literally. It is part of the language construct.

4.2. Running I/O

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 *I/O operations cannot be encapsulated using* `runST`. Why not? Again, because of `runST`'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 I/O 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.

How, then, are I/O operations executed at all? The meaning of the whole program is given by the value of the top-level identifier `mainIO`:

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

`mainIO` is an I/O state transformer, which is applied to the external world state by the operating system. Semantically speaking, it returns a new world state, and the changes embodied therein are applied to the real world. Of course, as in the case of mutable variables, our intention is that the program will actually mutate the real world "in place".

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 the X window system, an image-processing library, and so on.

5. Some applications

In this section we give three applications of the state transformer framework, and assess its usefulness.

5.1. A functional `more`

Our first example concerns input/output. The Unix `more` utility allows the user to view the contents of a file, a screenful at a time. A status line is displayed to give positional information, and commands are provided to allow scrolling in either direction.

However, `more` can only be used to display ASCII text. A completely different program would be needed to scroll through a file of pictures, or to scroll and render a file of PostScript. We can improve on this situation by separating the process into two:

- A representation-specific function, which converts the ASCII file, description of pictures, PostScript, or whatever, to a list of I/O actions, each of which paints a single screenful:

  ```
  asciiPages      :: String -> [IO ()]
  picturePages    :: PictureDescriptions -> [IO ()]
  postscriptPages :: String -> [IO ()]
  ```

- A representation-independent function, which takes a list of I/O actions (one for each screenful), and interacts with the user to navigate through them:

  ```
  more :: [IO ()] -> IO ()
  ```

Now the three viewers can be built by composition:

```
moreAscii      = more . asciiPages
morePictures   = more . picturePages
morePostscript = more . postscriptPages
```

Notice the complete separation of concerns. The `asciiPages` function is concerned only with displaying ASCII text on the screen, and not at all with user interaction. The `more` function is concerned only with interaction with the user, and not at all with the nature of the material being displayed. This separation of concerns is achieved by passing a list of I/O actions from one function to another. Notice too that some of these actions may be performed more than once if, for example, the user scrolls back to a previously-displayed page. None of this is possible in any imperative language we know of.

Our solution has its shortcomings. The Unix `more` allows the user to scroll a single line at a time, whereas ours only allows scrolling in units of pages. It is a common discovery that good abstractions sometimes conflict with arbitrary functionality!

5.2. Depth first search

Depth-first search is a key component of many graph algorithms. It is one of the very few algorithms for which an efficient algorithm is most lucidly expressed using mutable state, so it makes a good application of the ideas presented in this paper.

Our depth-first search function, `dfs`, is given in Figure 2. It takes a graph `g` and a list of vertices `vs`, and returns a list of trees — or forest — which, collectively, span `g`. Furthermore, the forest has the "depth-first" property; that is, no edge in the original graph traverses the forest from left to right. The list of vertices `vs` gives an initial ordering for searching the vertices, which is used to resume the search whenever one is completed. Clearly the head of `vs` will be the root of the very first tree.

The graph is represented by an array, indexed by vertices, in which each element contains a list of the vertices reachable directly from that vertex. The `dfs` function begins by introducing a fresh state thread, allocating an array of marks initialised to `False`, and then calling the locally defined function `search`. The whole thing is encapsulated by `runST`.

When searching a list of vertices, the mark associated with the first vertex is examined, and if `True` the vertex is discarded and the rest are searched. If however the mark is `False` indicating that the vertex has not been examined previously, then it is marked `True`, and two recursive calls of search are performed, each of which returns a list of trees. The first call, namely, `search marks (g!v)`, is given the edges leading from `v`, and it produces a forest `ts` which is built into a tree with `v` at the root—all these nodes are reachable from `v`. The second recursive call (`search marks 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.

In a non-strict language, an expression is evaluated in response to demands from the consumer of the expression's value. This property extends to values produced by stateful computations. In the case of depth-first search, if only part of the forest returned by `dfs` is evaluated then only part of the stateful computation will be carried out. This is quite a remarkable property: we know of no other system

```
type Graph  = Array Vertex [Vertex]
data Tree a = Node a [Tree a]

dfs :: Graph -> [Vertex] -> [Tree Vertex]
dfs g vs = runST (
              newArr (bounds g) False `thenST` \ marks ->
              search marks vs
           )

  where search :: MutArr s Vertex Bool -> [Vertex]
                -> ST s [Tree Vertex]
        search marks   []   = returnST []
        search marks (v:vs) = readArr marks v `thenST` \ visited ->
                              if visited then
                                 search marks vs
                              else
                                writeArr marks v True  `thenST_`
                                search marks (g!v)     `thenST` \ ts ->
                                search marks vs        `thenST` \ us ->
                                returnST ((Node v ts): us)
```

Figure 2. Lazy depth-first search

which can execute a sequential, imperative algorithm, incrementally in response to demands on its result value.

This algorithm, and many others derivable from it, or definable in terms of it are discussed in detail in King & Launchbury [1993].

5.3. An interpreter

We conclude this section with a larger example of array references in use (Figure 3). 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[6].

The resulting program has the same laziness property as our depth-first search. As successive elements of the result of a call to `interpret` are evaluated, the interpreter will incrementally execute the program just far enough to get to the next `Write` command, when the `returnST` delivers a new element of the result list, and no further. If only the first few elements of the result are needed, much of the imperative program being interpreted will never be executed at all.

```
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 3. An interpreter with lazy stream output

This example has long been a classic test case for systems which infer single-threadedness (Schmidt [1985]), and is also used by Wadler in his paper on monads (Wadler [1992a]). 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.

5.4. Summary

An obvious question arises when looking at monadic code: it appears to differ only superficially from an ordinary imperative program. Have we done any more than discover a way to mimic C in Haskell?

We believe that, on the contrary, there are very significant differences between writing programs in C and writing in Haskell with monadic state transformers and I/O:

- Usually, most of the program is neither stateful nor directly concerned with I/O. The monadic approach allows the graceful co-existence of a small amount of "imperative" code and the large purely functional part of the program.

- The "imperative" component of the program still enjoys all the benefits of higher order functions, polymorphic typing, and automatically-managed storage.

- A state transformer corresponds, more or less, to a statement in C or Pascal[7]. However, a state transformer is a first-class value, which can be stored in a data structure, passed to a function, returned as a result, and so on, while a C statement enjoys none of these properties. This expressive power was used in the `more` example above (Section 5.1), where complete I/O actions are constructed by one function, stored in a list, and subsequently performed, perhaps repeatedly, under the control of an entirely separate function.

- Imperative languages provide a fixed repetoire of control structures (conditional statements, while loops, for loops, and so on). Because state transformers are first class values, the programmer can define functions which play the role of *application-specific* control structures. For example, here is a function which performs its argument a specified number of times:

  ```
  repeatST :: Int -> ST s () -> ST s ()
  repeatST n st = listST_ [st | i <- [1..n]]
  ```

  If we need a `for` loop, where the loop index is used in the "body", it is easily provided:

  ```
  forST :: Int -> (Int -> ST s ()) -> ST s ()
  forST n stf = mapST_ stf [1..n]
  ```

And so on. The point is not that these particular choices are the "right" ones, but rather that it is very easy to define new ways to compose together small state transformers to make larger ones.

- The usual co-routining behaviour of lazy evaluation, in which the consumer of a data structure co-routines with its producer, extends to stateful computation as well. As Hughes argues (Hughes [1989]), the ability to separate *what* is computed from *how much* of it is computed is a powerful aid to writing modular programs.

6. Formalism

Having given the programmer's eye view, it is time now to be more formal and to define precisely the ideas we have discussed. We have presented state transformers in the context of the full-sized programming language Haskell, since that is where we have implemented the ideas. In order to give semantics to the constructs, however, it is convenient to restrict ourselves to the essentials. In particular, we choose to omit the special semantics of IO operations with their calls to C. Instead, we focus on providing a semantics for encapsulated state together with a proof demonstrating the security of encapsulation.

6.1. A Language

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

$$\begin{array}{rcl} e & ::= & x \mid k \mid e_1\, e_2 \mid \lambda x.e \mid \\ & & \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \mid \texttt{runST}\ e \\ \\ k & ::= & \texttt{thenST} \mid \texttt{returnST} \mid \\ & & \texttt{newVar} \mid \texttt{readVar} \mid \texttt{writeVar} \mid \\ & & \ldots \end{array}$$

6.2. Type rules

The type rules are given in Figure 4, and are the usual Hindley-Milner rules, except that `runST` also requires a judgment of its own. 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 provide a typing judgement which has the same effect.

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

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

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

$$VAR \qquad \Gamma, x : S \vdash x : S$$

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

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

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

Figure 4. Type rules

$$
\begin{array}{lcl}
\mathcal{E}[\![\, Expr \,]\!] : Env \to Val & & \\
 & & \\
\mathcal{E}[\![\, k \,]\!]\ \rho & = & \mathcal{B}[\![\, k \,]\!] \\
\mathcal{E}[\![\, x \,]\!]\ \rho & = & \rho\ x \\
\mathcal{E}[\![\, e_1\ e_2 \,]\!] & = & (\mathcal{E}[\![\, e_1 \,]\!]\ \rho)\ (\mathcal{E}[\![\, e_2 \,]\!]\ \rho) \\
\mathcal{E}[\![\, \backslash x\texttt{->}e \,]\!]\ \rho & = & \lambda v.(\mathcal{E}[\![\, e \,]\!]\ (\rho \oplus \{x \mapsto v\})) \\
\mathcal{E}[\![\, \texttt{let}\ x\texttt{=}e_1\ \texttt{in}\ e_2 \,]\!]\ \rho & = & \mathcal{E}[\![\, e_2 \,]\!]\ (fix(\lambda\rho'.(\rho \oplus \{x \mapsto \mathcal{E}[\![\, e_1 \,]\!]\rho'\}))) \\
 & & \\
\mathcal{B}[\![\, \texttt{runST} \,]\!] & = & runST \\
\mathcal{B}[\![\, \texttt{thenST} \,]\!] & = & thenST \\
\vdots & &
\end{array}
$$

Figure 5. Semantics of Terms

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 $Bool$.

$$
\begin{array}{rcl}
T & ::= & t \mid K \mid T_1 \to T_2 \mid \\
 & & \texttt{ST}\ T_1\ T_2 \mid \\
 & & \texttt{MutVar}\ T_1\ T_2 \\
 & & \\
S & ::= & T \mid \forall t.S
\end{array}
$$

In addition, Γ ranges over type environments (that is, partial functions from term variables to types), and we write $FV(T)$ for the free variables of type T and likewise for type environments.

6.3. Denotational Semantics

In Figure 5, we extend a standard denotational semantics for a non-strict lambda calculus to include the semantics of state operations by providing definitions for the new constants.

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:

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

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 (polymorphic values lie in the intersection of their monomorphic instances).

$$
\begin{array}{lcl}
(\mathit{returnST}\ v)\ \sigma & = & (v, \sigma) \\
(\mathit{thenST}\ m\ k)\ \sigma & = & k\ x\ \sigma'\ \textbf{where}\ (x, \sigma') = m\ \sigma
\end{array}
$$

$$
\mathit{newVar}\ v\ \sigma \quad = \begin{cases} (\bot,\ \bot), & \textbf{if } \sigma = \bot \\ (p,\ \sigma[p \mapsto v]), & \textbf{otherwise} \end{cases} \\
\qquad\qquad \textbf{where } p \notin \mathit{dom}(\sigma)
$$

$$
\mathit{readVar}\ p\ \sigma \quad = \begin{cases} (\bot,\ \bot), & p \notin \mathit{dom}(\sigma) \\ (\sigma\ p,\ \sigma), & \textbf{otherwise} \end{cases}
$$

$$
\mathit{writeVar}\ p\ v\ \sigma \ = \begin{cases} (\bot,\ \bot), & p \notin \mathit{dom}(\sigma) \\ ((),\ \sigma[p \mapsto v]), & \textbf{otherwise} \end{cases}
$$

$$
\mathit{runST}\ m \ = \ x \ \ \textbf{where}\ (x, \sigma) = m\ \emptyset
$$

Figure 6. Semantics of State Combinators and Primitives

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

$$
\begin{array}{lcl}
D_{\mathtt{ST}\ s\ a} & = & \mathit{State} \to (D_a \times \mathit{State}) \\
\mathit{State} & = & (\mathcal{N} \hookrightarrow \mathit{Val})_\bot \\
D_{\mathtt{MutVar}\ s\ a} & = & \mathcal{N}_\bot
\end{array}
$$

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 the bottom value and the bottom state.

A state is a lifted finite partial function from locations (represented by natural numbers) to values. The bottom state is totally undefined. This is the state that results after an infinite loop. We cannot tell even which variables exist, let alone what their values are. Non-bottom states are partial functions with well-defined domains which specify which variables exist. These variables may be mapped to any value, including bottom.

References are denoted simply by natural numbers, except that it is possible to have an undefined reference also, denoted by $\bot$. The number represents a "location" in the state.

It is worth noting in passing that the state parameter s is ignored in providing semantic meaning. It is purely a technical device which shows up in the proofs of safety.

The definitions of the state constants are given in Figure 6. Neither `returnST` nor `thenST` are strict in the state, but both are single threaded. Thus they sequentialise state operations, and guarantee that only one copy of the state is required, without themselves forcing operations to be performed.

$$newVars_S\ v\ \sigma \quad = \begin{cases} (\bot,\ \bot), & \textbf{if } \sigma = \bot \\ (\theta_S(p),\ \sigma[p \mapsto v]), & \textbf{otherwise} \end{cases}$$
$$\textbf{where } p \notin dom(\sigma)$$

$$readVars_S\ q\ \sigma \quad = \begin{cases} (\bot,\ \bot), & p \notin dom(\sigma) \\ (\sigma\ (\theta_S^{-1}\ (q)),\ \sigma), & \textbf{otherwise} \end{cases}$$

$$writeVars_S\ q\ v\ \sigma \ = \begin{cases} (\bot,\ \bot), & p \notin dom(\sigma) \\ ((),\ \sigma[\theta_S^{-1}(q), \mapsto v]), & \textbf{otherwise} \end{cases}$$

$$\{\theta_\tau : \mathcal{N} \to \mathcal{N} \mid \tau \in Type, \theta_\tau\ 1 - to - 1\}$$

$$D_{\texttt{MutVar}\ s\ a} = (ran\ (\theta_S))_\bot$$

Figure 7. Semantics of Indexed State Primitives

In contrast, the primitive operations are strict in the state. In order to allocate a new variable, for example, we need to know which locations are free. Similarly, with the other operations (in the semantics, we take $p \notin dom(\sigma)$ to be true if either p or σ are bottom). This all has a direct operational reading which we discuss in Section 9.3.3.

Finally, the meaning of `runST` is given by applying its state-transforming argument to the empty state, and extracting the first component of the result. Note that `runST` is not strict in the final state—it is simply discarded.

6.4. Safety

In any program there may be lots of concurrently active state threads which must not be allowed to interfere with one another. That is state changes performed by one thread must not affect the result of any other.

We achieve this by using the type scheme for `runST`. We use the parametric nature of polymorphism to show that values within a state thread (including the final value) cannot depend on references generated by other state threads. Section 6.4.2 contains the proof of the key lemma. Here, we will focus on the overall reasoning process, assuming that the lemma holds.

6.4.1. Independence of threads

Figure 7 contains a more complex version of the semantics of the state primitives in which the references are respectively coded and decoded. We assume an indexed family of injections θ from the naturals to themselves to act as codings. This family is indexed by types to allow the different instances of the primitives to use different encodings for the variables (because the coding function is selected on the basis of the particular instance type).

When $newVar$ allocates a new location, rather than return the address of the location, it codes it using a particular coding function θ. When $readVar$ and $writeVar$ come to dereference the variable, they first apply the inverse of the coding function to find the true location, and then act as before.

All the component parts of the state thread are eventually joined together using `thenST` whose type forces these various parts to have the same state instance, and hence use the same coding function. Similarly, the fact that references carry the type of their creator forces any dereferencer to be in the same instance, and also use the same coding function. Intuitively, therefore, the behaviour of a state thread should be independent of the choice of coding function. Only if references managed to cross state boundaries, or if the state information could somehow become lost in the process of typechecking, could something go wrong.

The essence of the proof is that nothing does go wrong. The key lemma states that a state transformer which is polymorphic in the state is indeed independent of the coding function used. This means that state references do not cross state boundaries, that is, no reference is dereferenced in any thread other than its creator. If the converse were so, then we could construct a state thread whose behaviour was *dependent* in the particular coding function used (change the coding function, so effectively changing what the external reference points to), contradicting the lemma.

Finally, as the choice of θ is irrelevant, we are free to use the identity function, and dispense with codings completely. This gives the original semantics of Figure 6.

6.4.2. Key Lemma

The development here relies on parametric polymorphism in the style of Mitchell & Meyer [1985], who detail what effect the addition of polymorphic constants has on parametricity. In general, we cannot hope that adding new polymorphic constants willy-nilly will leave parametricity intact. In particular, the addition of a "polymorphic equality" will have a disastrous effect on the level of parametricity implied by a polymorphic type.

Mitchell and Meyer show that parametricity is preserved so long as (1) each new type constructor comes equipped with a corresponding operation on relations, and (2) the new polymorphic constants all satisfy the relevant logical relations implied by their types.

In our setting, this means that we have to give an account of what `State` and `MutVar` do to relations, and show that `newVar`, `readVar` and `writeVar` satisfy the logical relations corresponding to their types.

Because we are here concerned purely with the issue of separating distinct state threads, we only focus on the polymorphism with repect to the state parameter.

Consequently, we will assume that we only store values of some fixed type, X say, in the state, and we completely sidestep the issue of proving that the retrieved values are of the same type as the `MutVar` reference which pointed to the location.

We define

$$Var\ s \ = \ MutVar\ a\ X$$

Thus `Var` is a unary type constructor which we now use instead of `MutVar`.

Let R and S be types, and let $r : R \leftrightarrow S$ be a relation[8] between R and S. Let q_R be a reference of type $Var\ R$ and q_S be type $Var\ S$. We define

$$\begin{array}{rcl} q_R(Var\ r)q_S & \equiv & \theta_R^{-1}(q_R) = \theta_S^{-1}(q_S) \\ \sigma_R(State\ r)\sigma_S & \equiv & \sigma_R = \sigma_S \end{array}$$

Two variables are related if they point to the same location, and two states are related if they are equal.

Lemma

`newVar`, `readVar` and `writeVar` are all logical relations.

Proof

We do the proof in the case of `newVar`. The others are just as easy. From the type of `newVar`, we have to show that for all $r : R \leftrightarrow S$,

$$newVar_R\ a\ \sigma(Var\ r \times State)newVar_S\ a\ \sigma$$

Expanding out the definition gives,

$$\theta_R^{-1}(\theta_R(p)) = \theta_S^{-1}(\theta_S(p)) \wedge \sigma[p \mapsto v] = \sigma[p \mapsto v]$$

which is clearly true. □

As the constants are parametric, so are terms built from them (this is the force of Mitchell and Meyer's result). Thus we deduce the key lemma as a corollary:

Lemma

If $m : \forall s.ST\ s\ T$ (where $s \notin FV(T)$) then for any types R and S, and any state σ we have,

$$m_R\sigma = m_S\sigma$$

As it is the instance of m which selects its coding for variables, the theorem states that the result of a polymorphic state transformer is independent of its internal coding. This is exactly what we needed to show for our earlier reasoning to be supported.

7. Haskell Arrays

Next, we turn our attention to the implementation of immutable Haskell arrays. We will show that the provision of *mutable* arrays (Section 3) provides an elegant route to an efficient implementation of *immutable* arrays.

7.1. Implementing array

As we have already indicated, Haskell's immutable arrays are constructed by applying the function `array` to a list of index-value pairs. For example, here is an array in which each element is the square of its index:

```
array (1,n) [(i,i*i) | i <- [1..n]]
```

This expression makes use of a *list comprehension* `[(i,i*i) | i <- [1..n]]`, which should be read "the list of all pairs `(i,i*i)` such that `i` is drawn from the list `[1..n]`. Now, it is obviously a big waste to construct the intermediate list of index value pairs! It would be much better to compile this expression into a simple loop which appropriately initialises each element of the array. Unfortunately, it is much less obvious how to achieve optimisation, at least in a way which is not "brittle". For example, it would be a pity if the optimisation was lost if the expression was instead written in this equivalent form:

```
array (1,n) (map (\i -> (i,i*i)) [1..n])
```

An obvious idea is to try to make use of *deforestation.* Deforestation is the generic name used for transformations which aim to eliminate intermediate lists — that is, lists used simply as "glue" between two parts of a functional program. For example, in the expression

```
map f (map g xs)
```

there is an intermediate list `(map g xs)` which can usefully be eliminated. Quite a bit of work has been done on deforestation (Chin [1990]; Marlow & Wadler [1993]; Wadler [1990]), and our compiler includes deforestation as a standard transformation (Gill, Launchbury & Peyton Jones [1993]).

The difficulty with applying deforestation to the construction of arrays is this: *so long as* `array` *is a primitive, opaque operation, there is nothing deforestation can do, because deforestation inherently consists of melding together part of the producer of a list with part of its consumer.* The flip side is this: if we can express `array` in Haskell, then our standard deforestation technique may be able to eliminate the intermediate list.

With this motivation in mind, we now give a definition of `array` using mutable arrays:

```
array :: Ix i => (i,i) -> [(i,v)] -> Array i v
array bds ivs
  = runST (
      newArr bds unInit      `thenST` \ arr ->
      mapST_ (fill arr) ivs  `thenST_`
      freezeArr arr
    )
  where
```

```
    unInit = error "Uninitialised element"
    fill arr (i,v) = writeArr arr i v
```

The definition can be read thus:

1. The call to `newArr` allocates a suitably-sized block of memory.

2. The call to `mapST_` performs in sequence the actions `(fill arr (i,v))`, for each `(i,v)` in the index-value list. Each of these actions fills in one element of the array.

3. The function `freezeArray` is a new primitive which converts the mutable array into an immutable array:

   ```
   freezeArr :: Ix i => MutArr s i v -> ST s (Array i v)
   ```

   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 9.3.4).

4. Finally, the whole sequence is encapsulated in a `runST`. Notice the use of encapsulation here. The *implementation* (or internal details) of `array` is imperative, but its *specification* (or external behaviour) is purely functional. Even the presence of state cannot be detected outside `array`.

The important thing is that the list of index-value pairs, `ivs` is now explicitly consumed by `mapST_`, which gives enough leverage for our deforestation transformation to eliminate the intermediate list. The details of the deforestation transformation are given in Gill, Launchbury & Peyton Jones [1993], and are not germane here. The point is simply that exposing the implementation of `array` to the transformation system is the key step.

7.2. Accumulating arrays

Of course, we can also define other Haskell array "primitives" in a similar fashion. For example, `accumArray` is a standard Haskell array operation with type:

```
  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` simple, and very similar to that of `array`:

```
accumArray bnds f z ivs
  = runST (
      newArr bnds z           `thenST` \a ->
      mapST_ (update a) ivs   `thenST_`
      freezeArr a
    )
  where
    update a (i,v) = readArr a i `thenST` \x->
                     writeArr a i (f x v)
```

If `array` and `accumArray` were primitive then the programmer would have no recourse if he or she wanted some other array-construction operator. Mutable arrays allow the programmer to define new array operations without modifying the compiler or runtime system. We describe a more complex application of the same idea in Section 10.3.

8. 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.

8.1. 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 *write*s to the one will affect *read*s from the other). It turns out to be quite straightforward to add an additional constant, `eqMutVar`:

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

8.2. Fixpoint

In lazy functional programs it is often useful to write recursive definitions such as

```
(newTree, min) = f tree min
```

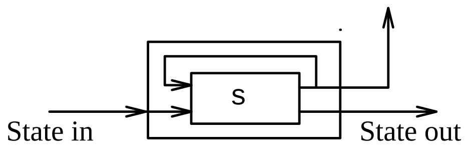
Here, part of the result of `f` is passed into `f`. A standard example is a function which replaces all the leaves of a tree with the minimum of all the leaves, in a single pass of the tree (hence our choice of names in the example) (Bird [1984]). The alert reader may have noticed that it is impossible to write programs in this way if `f` is a state transformer. We might try:

```
f tree min  ‘thenST‘ \ (newTree, min) -> ...
```

but alas the `min` result is not in scope at the call. What is needed is a new combinator `fixST`, with type:

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

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



Now we can write our recursive state transformer:

```
fixST (\ ~(newTree, min) -> f tree min)
```

(The "~" in this example specifies that the tuple should be matched lazily. If the argument to `fixST` is a strict function, then of course the result of the `fixST` call is bottom.)

9. 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. This section focuses on implementation issues, and appeals to some intuitions about what will generate "good code" and what will not. Readers interested in a more substantial treatment of such intuitions are referred to Peyton Jones [1987], Peyton Jones [1992].

We have in mind the following 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.

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.

9.1. 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 neither duplicate nor discard it (Figure 5). Furthermore, all the primitive operations are strict in the state.

It follows that 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, and the preceding operation will not produce its result state until it has computed its result, there can be no other as-yet-unexecuted 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` and `returnST` and `fixST`. 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.

9.2. Typechecking `runST`

Since `runST` has a rank-2 type, we needed to modify the type checker to include the extra rule RUN in Figure 4. The modification is quite straightforward, because the rule for RUN is so similar to that for LET. All that is required is to check that the type inferred for the argument of `runST` has a type of the form `ST` s τ, where s is an un-constrained type variable, not appearing in τ.

The RUN rule describes how to type-checking *applications* of `runST`. What of other occurrences of `runST`? For example, the RUN rule does not say how to type the following expression, in which `runST` appears as an argument:

```
map runST xs
```

We side-step this difficulty by insisting that `runST` only appears applied to an argument, as implied by the syntax in Section 6.1. It might be possible to allow greater generality, but performing type inference in the presence of unrestricted rank-2 types is a much harder proposition, and one which is not necessary for our enterprise. Kfoury [1992] and Kfoury & Wells [1994] explore this territory in detail.

9.3. Efficiency considerations

It would be possible to implement state transformers by providing the combinators (`thenST`, `returnST`, etc) and operations (`readVar`, `writeVar` etc) as primitives. 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 in the heap 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!

We might attempt to recover some of this lost efficiency by treating 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 state transformer type and its combinators. These definitions are almost precise transliterations of the semantics given for them in Figure 5[9].

9.3.1. The state transformer implementation

A state transformer, of type `ST s a`, is represented by a function from `State s` to a pair of the result, of type `a` and the transformed state.

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

(This representation of `ST` is not, of course, exposed to the programmer, lest he or she write functions which duplicate or discard the state.) The one respect in which this implementation differs from the semantics in Figure 5 is in the result type of `ST`: the semantics used a simple product, whereas a Haskell pair is a lifted product. (That is, the value $\perp$ differs from ($\perp$,$\perp$).) As we shall see, this distinction has an unfortunate implication for efficiency.

The definitions of `thenST`, `returnST`, and `fixST` follow immediately:

```
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
```

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`, the much more efficient expression

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

We have not so far given the implementation of `runST`, which is intriguing:

```
runST m = r where (r,s) = m dummyState
```

Since its argument, `m`, works regardless of what state is passed to it, we simply pass a value representing the current state of the world. As we will see shortly (Section 9.3.3), this value is never actually looked at, so a constant value, `dummyState`, will do. We need to take care here, though. Consider the following expression, which has two distinct state threads:

```
...(runST (newVar 1 `thenST` \v -> e1))...
...(runST (newVar 1 `thenST` \v -> e2))...
```

After inlining runST and thenST we transform to:

```
...(let (v,s1) = newVar 1 dummyState in e1 s1)...
...(let (v,s1) = newVar 1 dummyState in e2 s1)...
```

Now, it look as though it would legitimate to share newVar 1 dummyState as a common sub-expression:

```
let (v,s1) = newVar 1 dummyState
in
...(e1 s1)...
...(e2 s1)...
```

but of course this transformation is bogus. The two dummyStates are distinct values! There are two solutions to this problem: do not inline runST, or alternatively, when inlining runST create a new constant dummyState on each occasion, akin to skolemization of logic variables.

This provides us with a second reason why runST should not be regarded as a standard value (its type provided the first). Rather, runST is an eliminable language construct.

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.

9.3.2. Strictness

In the previous section we produced the following code from a composition of m1 and m2:

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

This might be better than the original, in which function-valued arguments are passed to thenST, but it is still not very good! In particular, heap-allocated thunks are created for m1 s and m2 s1, along with heap-allocated selectors to extract the components (v1, s1 and v2, s2, respectively). However, the program is now exposed to the full range of analyses and program transformations implemented by the compiler. 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)
```

This is *much* more efficient. First `m1` is called, returning a pair which is taken apart; then `m2` is called, and its result taken apart before returning the final result. In our implementation, no heap allocation is performed at all. If `m1` and `m2` are primitive operations, then the code implementing `m2` simply follows that for `m2`, just as it would in C.

It turns out that even relatively simple strictness analysis can often successfully enable this transformation, provided that Haskell is extended in a modest but important regard. Consider the function

```
f :: MutVar s Int -> ST s ()
f x = writeVar y 0 `thenST_`
      returnST ()
```

Is `f` strict in its input state? Intuitively, it must be: its only useful result is its result state, which depends on its input state. This argument holds for any value of type `ST s ()`. Does the strictness analyser spot this strictness? Alas, it does not. After inlining, `f` becomes:

```
f x s = let (_,s1) = writeVar y 0 s
        in
        ((),s1)
```

This definition is manifestly not strict in either argument: it will return a pair regardless of the values of its arguments. There are two problems, with a common cause:

- Haskell pairs are lifted, so that $(\perp,\perp)$ is distinct from $\perp$. The semantics in Figure 5 used an ordinary, unlifted product, so the lifting is not certainly not required by semantics.
- The unit type, `()`, is also a lifted domain, with two values: `()` and $\perp$. When we chose `()` as the result type of state transfomers which had no result we certainly did not have in mind that two distinct values could be returned. Again, the lifting of the Haskell type `()` is not required.

If neither pairs nor the unit type were lifted then it is easy to see that `f` is strict in `s`. Suppose `s` were $\perp$: then because the primitive `writeVar` is strict, `s1` would also be $\perp$, so the result of `f` would be the pair $((),\perp)$. But if `()` were unlifted too, this is the same as $(\perp,\perp)$, which, if pairs are unlifted, is the same as $\perp$.

In short, in order to give the strictness analyser a real chance, a state transformer must return an unlifted pair, and the result type of a state transformer which is used only for its effect on the state should be an unlifted, one-point type. Though this is an important effect, it is far from obvious; indeed, it only became clear to us as we were working on the final version of this paper.

It is also regrettable that an occasionally-substantial performance effect should depend on something as complex as strictness analysis. Indeed, we provide a variant of `returnST`, called `returnStrictlyST`, which is strict in the state, precisely to allow a programmer to enforce strictness, and hence ensure greater efficiency. Of course, if `returnStrictlyST` is used indiscriminately then the incremental laziness of stateful computations (discussed in Section 5) is lost.

In the special (but common) case of I/O state transformers, we can guarantee to compile efficient code, because *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 change to the real world! It is easy to use this strictness property (which cannot, of course, be inferred by the strictness analyser) to ensure that every I/O state transformer is compiled efficiently.

9.3.3. 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 state changes in the common, global heap. What, then, is the role of the explicit state value which is 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#`.

9.3.4. 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 `unsafeFreezeArray`, 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 `unsafeFreezeArray`.

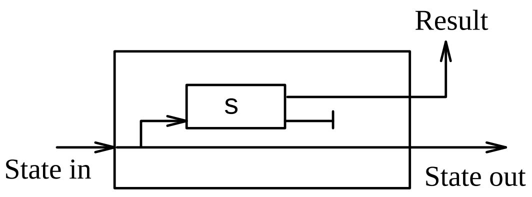
10. Interleaved state transformers

The state-transformer composition combinator defined so far, `thenST`, is completely 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
interleaveST m s = (r,s)  where (r,s') = m s
```

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



The semantics of `interleaveST` are much less easy to define and reason about than the semantics of our other combinators, given that our intended implementation remains that of update in place. The purpose of this section is to provide several motivating examples for `interleaveST`, while leaving open the question of how to reason about whether it is being used "safely". In this context, we regard a use of `interleaveST` as "safe" if the resulting program can still be evaluated in any order that respects data dependencies.

10.1. Lazy file read

One way for `interleaveST` to be safe is to regard it as splitting the state into two disjoint parts. 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.

Here is an implementation of lazy file read, using `interleaveST`:

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

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

Notice that the recursive call to `readCts` does not immediately read the rest of the file. Because `thenST` and the following `returnST` are non-strict, the list `c:cs` will be returned without further ado. If `cs` is ever evaluated, the recursive `readCts` will then (and only then) be performed. This is a good example of laziness in action. Even though operations are being rigidly sequenced — in this case the reads of successive characters of the file — the rate at which this sequence is performed is driven entirely by lazy evaluation. The single call to `interleaveST` simply allows these operations to occur asynchronously with respect to other I/O operations on the main "trunk".

10.2. Unique-supply trees

In the lazy file-read example, the use of `interleaveST` could be regarded as splitting the state into two disjoint parts. However, we have found some compelling situations in which the forked-off thread quite deliberately shares state with the "trunk". This section explores the first such example.

A common problem in functional programs is to distribute a supply of unique names around a program. For example, a compiler may want to give a unique name to each identifier in the program being compiled. In an imperative program one might simply call `GenSym()` for each identifier, to allocate a unique name from a global supply, and to side-effect the supply so that subsequent calls to `GenSym()` will deliver a new value.

In a functional program matters are not so simple; indeed, several papers have discussed the problem (Augustsson, Rittri & Synek [1994]; Hancock [1987]; Wadler [1992a]). The rest of this section shows a rather elegant implementation of the best approach, that of Augustsson, Rittri & Synek [1994]. The idea is to implement a pair of abstract data types, `UniqueSupply` and `Unique`, with the following signature:

```
newUniqueSupply   :: IO UniqueSupply
splitUniqueSupply :: UniqueSupply -> (UniqueSupply, UniqueSupply)
getUnique         :: UniqueSupply -> Unique

instance Eq   Unique
instance Ord  Unique
instance Text Unique
```

The three `instance` declarations say that the `Unique` type has defined on it equality and ordering operations, and mappings to and from strings. Naturally, the idea is that the two `UniqueSupply`s returned by `splitUniqueSupply` are forever separate, and can never deliver the same `Unique`. The implementation is given some extra freedom by making `newUniqueSupply` into an I/O operation. Different runs of the same program are therefore permitted to allocate uniques in a different order — all that matters about `Unique`s is that they are distinct from each other.

One possible implementation would represent a `UniqueSupply` and a `Unique` by a (potentially very long) bit-string. The `splitUnique` function would split a supply into two by appending a zero and a one to it respectively. The trouble is, of course, that the name supply is used very sparsely.

The idea suggested by Augustsson, Rittri & Synek [1994] is to represent a `UniqueSupply` by an infinite tree, which has a `Unique` at every node, and two child `UniqueSupply`s:

```
data UniqueSupply = US Unique UniqueSupply UniqueSupply
```

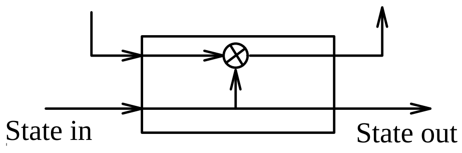
Now the implementation of `splitUniqueSupply` and `getUnique` are trivial, and all the excitement is in `newUniqueSupply`. Here is its definition, assuming for the sake of simplicity that a `Unique` is represented by an `Int`:

```
type Unique = Int

newUniqueSupply :: IO UniqueSupply
newUniqueSupply
  = newVar 0          `thenST` \ uvar ->
    let
      next :: IO Unique
      next = interleaveST (
                readVar uvar          `thenST` \ u ->
                writeVar uvar (u+1)   `thenST_`
                returnStrictlyST u
              )

      supply :: IO UniqueSupply
      supply = interleaveST (
                 next         `thenST` \ u ->
                 supply       `thenST` \ s1 ->
                 supply       `thenST` \ s2 ->
                 returnST (US u s1 s2)
               )
    in
    supply
```

The two uses of `interleaveST` specify that the relative ordering of the state changes in `next`, and in the two uses of `supply`, is deliberately left to the implementation. The only side effects are in `next`, which allocates a new `Unique`, so what this amounts to is that the uniques are allocated in the order in which the `Unique`s are evaluated, which is just what we wanted!

If the program was recompiled with, say, a different analysis technique which meant that the evaluation order changed, then indeed different uniques would be generated. But since the whole `mkUniqueSupply` operation is typed as an I/O operation *there is no reason to suppose that the same uniques will be generated*, a nice touch.

There is one important subtlety in the code, namely the use of `returnStrictlyST` in the definition of `next`. The trap (into which we fell headlong) is this: *since* `interleaveST` *discards the final state, and the result of the* `writeVar` *is also discarded, if* `returnST` *is used instead there is nothing to force the* `writeVar` *to take place at all*! Indeed, if we used the standard `returnST`, no `writeVar`s would be performed, and each `readVar` would see the same, undisturbed value. Of course, what we want to happen is that once the thread in `next`'s right-hand-side is started, then it must run to completion. That is exactly what `returnStrictlyST` achieves. The difference between `returnST` and `returnStrictlyST` is simply that the latter is strict in the state. Operationally, it will not deliver a result at all until it has evaluated its input state. We can picture it like this (compare the picture for `returnST` in Section 2.1):



In this picture, the result is held up by the "valve" until the state is available.

Despite this subtlety, this code is a distinct improvement on that given in Augustsson, Rittri & Synek [1994], which has comments such as "The *gensym* function must be coded in assembler" and "A too-clever compiler might recognise the repeated calls to *gen x* and [generate] code which creates sharing...if such compiler optimisations cannot be turned off or fooled, one must generate code for *gen* [as well as *gensym*] by hand.".

10.3. Lazy arrays

A second example of the use of `interleaveST` where the forked-off thread shares state with the trunk concerns so-called *lazy arrays*. Haskell arrays are strict in the list of index-value pairs, and in the indices in this list (though not in the values). An alternative, and more powerful, array constructor would have the same type as `array`, but be non-strict in the index-value pairs:

```
  lazyArray :: Ix i => (i,i) -> [(i,v)] -> Array i v
```

What does it mean for `lazyArray` to be non-strict in the index-value pairs? Presumably, it must return an array immediately, and search the list only when the array is indexed. Since checking for duplicate indices would entail searching the whole list, `lazyArray` simply returns the value in the *first* index-value pair in the list with the specified index. In short, the semantics of indexing a lazy array is precisely that of searching an association list — except that, of course, we hope that it will be more efficient. Figure 8 gives a (rather amazing) program which uses lazy arrays to compute prime numbers using the sieve of Eratosthenes.

The basis of the algorithm is an array `minFactor` which, for each index, stores the minimum factor (¿1) of that index. A number is prime, therefore, if the value stored at its index in `minFactor` is equal to the number itself. The function `multiples` generates one or more multiples of its argument (each paired with its argument) by first returning its argument, and then, if the argument is prime, returning more multiples. Note that `multiple` always returns its argument as a multiple *before* checking its primality so guaranteeing that `minFactor` will have been initialised (if not by this call of `multiples`, then by a previous). This prevents the program from entering a black hole (deadlock).

How can we implement `lazyArray`? The effect we want to achieve is:

- `lazyArray` allocates and initialises a suitable array, and returns it immediately without looking at the index-value pairs.

```
primesUpTo :: Int -> [Int]
primesUpTo n = filter isPrime [2..n]
  where

    minFactor :: Array Int Int
    minFactor =  lazyArray (2,n) (concat (map multiples [2..n]))

    isPrime p :: Int -> Bool
    isPrime p =  minFactor!p == p

    multiples   :: Int -> [(Int,Int)]
    multiples k =  (k,k) : if isPrime k
                              then [(m,k) | m <- [2*k,3*k..n]]
                              else []
```

Figure 8. Computing primes using the Sieve of Eratosthenes, using a lazy array

- When this array is indexed, with the standard `(!)` operator, the index-value list is searched for the specified index. As a side effect of this search, the array is filled in with all the index-value pairs encountered, up to and including the index sought. When this index is found the search terminates, and the corresponding value is returned.
- If the array is subsequently indexed at the same index, the value is returned immediately.

This behaviour is not easy to achieve in Haskell! It relies inherently on imperative actions. The fact that the results are independent of the order in which array elements are accessed is a deep property of the process. Nevertheless, we can express it all with the help of `interleaveST`; the code is given in Figure 9.

Referring first to Figure 9, `lazyArray` allocates the following mutable values:

- A variable, `feederVar`, to hold the "feeder-list" of index-value pairs.
- A mutable array, `valArr`, in which the result of the whole call to `lazyArray` is accumulated.
- A mutable array of booleans, `doneArr`, whose purpose is to record when the corresponding slot of `valArr` has been assigned with its final value. Each element of the `doneArr` is initialise to `False`.

Next, `lazyArray` initialises each slot in `valArr` by calling `initVal`, defined in the `let`. Finally, `lazyArray` freezes the array and returns the frozen value. Here, we *must* use `unsafeFreezeArray` because the whole idea is that `valArr` is going to be mutated as a side effect of the evaluation of its elements. (Recall that

```
lazyArray :: Ix i => (i,i) -> [(i,val)] -> Array i val
lazyArray bounds feederList
  = runST (
      newVar feederList                 `thenST` \ feederVar ->
      newArray bounds False             `thenST` \ doneArr ->
      newArray bounds initial           `thenST` \ valArr ->

      let
        initVal k = interleaveST (
                       readVar feederVar          `thenST` \ ivs ->
                       writeVar feederVar badVal  `thenST_`
                       fillUntil k ivs            `thenST` \ ivs' ->
                       writeVar feederVar ivs'    `thenST_`
                       readArray valArr ix
                                  )         `thenST` \ delayedVal ->
                    writeArray valArr k delayedVal

        fillUntil k [] = noValue k
        fillUntil k ((i,v) : ivs)
          = readArray doneArr i           `thenST` \ done ->
            (if not done then
               writeArray doneArr i True   `thenST_`
               writeArray valArr i v
             else
               returnST () )              `thenST_`

            if i == k then
              returnST ivs
            else
              fillUntil k ivs
      in

      mapST initVal (range bounds)  `thenST_`
      unsafeFreezeArr valArr )

  where
    initial   = error "lazyArray: uninitialised element"
    badVal    = error "lazyArray: an index depends on a later value"
    noValue k = error ("lazyArray: no value for: " ++ show (index bounds k))
```

Figure 9. An implementation of lazy arrays

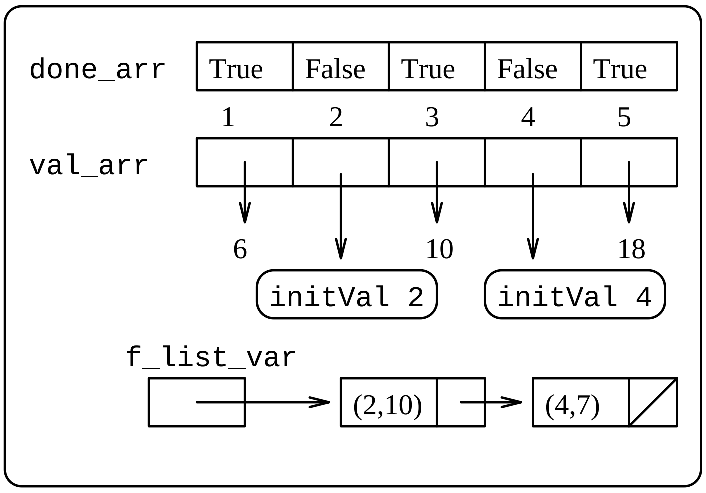


Figure 10. Lazy array construction

`unsafeFreezeArray` converts a mutable array to a value of the immutable-array type, without copying the array; further mutations of the mutable array will therefore also affect the "immutable" value.)

Referring now to the local functions: the function `initVal` initialises the slot of `valArr` with an interleaved state transformer which, when its result is demanded, will:

- read `feederVar` to get the current list of as-yet-unconsumed index-value pairs;
- call `fillUntil`, to consume the list of index-value pairs, writing each value to the appropriate slot of `valArr` (the array `done` records whether that element has already been written), until the sought-for index is found;
- write the depleted list of index-value pairs back to `feederVar`;
- read the array to deliver the desired value.

Further accesses to the same array slot will now find the final value, rather than the interleaved state transformer. All of this is illustrated by Figure 10, which illustrates an intermediate state of a lazy-array value. The array has five slots, of which three have already been filled in. The remaining two are specified by the depleted portion of the list of index-value pairs, contained in `feederVar`.

10.4. Parallel state transformers

A careful reading of `newUniqueSupply` or `lazyArray` reveals an important respect in which both are unsafe (in the sense of independence of evaluation order). Both rely on reading a variable and writing back a modified value — in the case of `newUniqueSupply` the variable is called `uvar`, while in `lazyArray` it is `feederVar`. Implicitly, we have assumed that if the read takes place then so will the write, and so they will in any sequential normal-order implementation. However, if two "threads" are executed simultaneously (which is one legitimate execution order) then this assumption might not hold, and the programs would fail.

This is not just a purist's objection, because an obvious development is a variant `interleaveST` which starts a concurrent process to execute the forked-off thread. Such a variant, which we call `forkST`, is very useful. For example, in a graphical I/O system it might be used to start a concurrent process to handle I/O in a new pop-up window, independent of, and concurrent with, other I/O operations.

In effect, `interleaveST` forces us to address the usual textbook problems of mutual exclusion and synchronisation that must be solved by any system supporting both concurrency and shared state. We are by no means the first to meet these issues in a functional setting. Concurrent ML (Reppy [1991]), and the I-structures (Arvind, Nikhil & Pingali [1989]) and M-structures (Barth, Nikhil & Arvind [1991]) of Id (Nikhil [1988]) are obvious examples. We are currently studying how to incorporate some of these now-standard solutions in our framework.

10.5. Summary

It should be clear by now that `interleaveST` has very undesirable properties. It duplicates and discards the state, which gives rise to a very subtle class of programming errors. We have so far failed to develop good techniques for reasoning about its correctness. At first we wondered about ways to ensure that the two state threads use different variables, but two of our most interesting applications, `lazyArray` and `newUniqueSupply`, deliberately perform side effects on *shared* state. Their correctness depends on (relatively) deep meta-reasoning, and a certain sort of atomicity (for example, the read and write of `uvar` must take place atomically in `next` in `newUniquesupply`).

Should we outlaw `interleaveST` on the grounds that it is insufficiently well behaved? Not necessarily. Outlawing `interleaveST` would simply drive its functionalitly underground rather than prevent it happening. For example, we want to have lazy file reading. If it cannot be implemented in Haskell then it will have to be implemented "underground" as a primitive operation written in C or machine code. The same goes for unique-supply trees and lazy arrays.

Implementing such operations in C does not make them more hygienic or easy to reason about. On the contrary, it is much easier to understand, modify and construct variants of them if they are implemented in a Haskell library module than if they are embedded in the runtime system.

The need for special care to be taken is flagged by the use of `interleaveST`, which identifies a proof obligation for the programmer to show that the results of the program are nevertheless independent of evaluation order. We fear that there may be no absolutely secure system — that is, one which guarantees the Church-Rosser property — which is also expressive enough to describe the programs which systems programmers (at least) want to write, such as those above. We do, however, regard `interleaveST` as useful primarily for systems programmers.

11. Related work

Monads were introduced to computer science by Moggi in the context of writing modular language semantics (Moggi [1989]). He noticed that denotational semantics of languages could be factored into two parts: what happens to values, and the underlying computational model. Furthermore, all the usual computational models satisfied the categorical definition of monads (often called *triples* in category theory) including state, exceptions, jumps, and so on. Wadler subsequently wrote several highly accessible papers showing how monads could be similarly used as a programming style in conventional functional programming languages (Wadler [1992a]; Wadler [1992b]; Wadler [1990]).

Based on this work, we developed a monadic approach to I/O (Peyton Jones & Wadler [1993]) and state (Launchbury [1993]) in the context of non-strict purely-functional languages. 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.

Both these shortcomings are solved in this paper, the core of which appeared earlier (Launchbury & Peyton Jones [1994]).

Several other languages from the functional stable provide some kind of state. For example, Standard ML provides *reference types*, which may be updated (Milner & Tofte [1990]). 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. In another way, though, effect systems are much more expressive than ours, because they provide a simple way to describe combinations of local bits of state. For example, one might have

$$\begin{array}{lcl} f & :: & a \rightarrow_{region_1} b \\ g & :: & c \rightarrow_{region_2} d \\ h & :: & e \rightarrow_{region_1 \cup region_2} f \end{array}$$

where h is defined using f and g. The side effects of f and g cannot interfere with each other (since they modify different regions) so they can proceed asynchronously with respect to each other.

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.

Our use of parametricity to encapsulate state has echoes of work by O'Hearn and Tennant (O'Hearn & Tennent [1993]). In imperative languages, local variables ought to be invisible to the the outside world. Most models of these languages, however, do not reflect this, making model-based reasoning about program behaviour less than satisfactory. O'Hearn and Tennant discovered that by using Reynolds-style relational parametricity, it was possible to construct models for these languages where local variables were truly invisible from outside the scope of the variable. One may view our work as extending this idea to the case where variables are first class values. Then it is no longer sufficient to restrict the application of parametricity solely at the model level — it becomes a source language concern instead.

Hall's recent work on list-compression in Haskell also has intriguing similarities to ours (Hall [1994]). Hall introduces an extra type variable into the list type, and uses its unification or otherwise to determine the scope of each list. If a particular application involving a list does not have the extra type variable instantiated to "unoptimised", then she replaces the list with a multi-headed version. This is rather similar to our use of universal quantification which, in essence, checks that the state-thread's state index is free of constraint.

It also seems possible to use existential quantification to achieve encapsulation of state. In classical logic, of course, a universal quantifier on the left of an arrow can be brought outwards, converting to an existential in the process. In intuitionistic logic, this is not valid in general. Nonetheless, Thiemann has developed a system which captures this (Thiemann [1993]), though the result is far more complex than the method presented here.

Acknowledgements

The idea of adding an extra type variable to distinguish 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`.

Philip Wadler made a number of insightful and helpful suggestions. In addition, all these ideas have benefited from discussions amongst the Functional Programming Group at Glasgow.

References

Arvind, RS Nikhil & KK Pingali [Oct 1989], "I-structures - data structures for parallel computing," *TOPLAS* 11, 598–632.

L Augustsson, M Rittri & D Synek [Jan 1994], "On generating unique names," *Journal of Functional Programming* 4, 117–123.

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.

PS Barth, RS Nikhil & Arvind [Sept 1991], "M-structures: extending a parallel, non-strict functional language with state," in *Functional Programming Languages and Computer Architecture, Boston*, Hughes, ed., LNCS 523, Springer Verlag, 538–568.

RS Bird [1984], "Using circular programs to eliminate multiple traversals of data," *Acta Informatica* 21, 239–250.

WN Chin [March 1990], "Automatic methods for program transformation," PhD thesis, Imperial College, London.

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

A Gill, J Launchbury & SL Peyton Jones [June 1993], "A short cut to deforestation," in *Proc Functional Programming Languages and Computer Architecture, Copenhagen*, ACM, 223–232.

CV Hall [June 1994], "Using Hindley-Milner type inference to optimise list representation," in *ACM Symposium on Lisp and Functional Programming, Orlando*, ACM, 162–172.

P Hancock [1987], "A type checker," in *The implementation of functional programming languages*, SL Peyton Jones, ed., Prentice Hall, 163–182.

John Hughes [Apr 1989], "Why functional programming matters," *The Computer Journal* 32, 98–107.

AJ Kfoury [June 1992], "Type reconstruction in finite rank fragments of second-order lambda calculus," *Information and Computation* 98, 228–257.

AJ Kfoury & JB Wells [June 1994], "A direct algorithm for type inference in the rank-2 fragment of the second-order lambda calculus," in *ACM Symposium on Lisp and Functional Programming, Orlando*, ACM, 196–207.

D King & J Launchbury [July 1993], "Functional graph algorithms with depth-first search," in *Glasgow Functional Programming Workshop, Ayr*.

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.

J Launchbury & SL Peyton Jones [June 1994], "Lazy functional state threads," in *SIGPLAN Symposium on Programming Language Design and Implementation (PLDI'94), Orlando*, ACM.

S Marlow & PL Wadler [1993], "Deforestation for higher-order functions," in *Functional Programming, Glasgow 1992*, J Launchbury & PM Sansom, eds., Workshops in Computing, Springer Verlag, 154–165.

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

R Milner & M Tofte [1990], *The definition of Standard ML*, MIT Press.

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

E Moggi [June 1989], "Computational lambda calculus and monads," in *Logic in Computer Science, California*, IEEE.

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

PW O'Hearn & RD Tennent [Jan 1993], "Relational parametricity and local variables," in *20th ACM Symposium on Principles of Programming Languages, Charleston*, ACM, 171–184.

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.

SL Peyton Jones [1987], *The Implementation of Functional Programming Languages*, Prentice Hall.

SL Peyton Jones [Apr 1992], "Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine," *Journal of Functional Programming* 2, 127–202.

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.

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

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.

P Thiemann [1993], "Safe Sequencing of Assignments in Purely Functional Programming Languages," Tech. Report. Wilhelm-Schickard-Institut WSI-93-i6, Tübingen, Germany.

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

PL Wadler [1990], "Deforestation: transforming programs to eliminate trees," *Theoretical Computer Science* 73, 231–248.

PL Wadler [1992a], "Comprehending monads," *Mathematical Structures in Computer Science* 2, 461–493.

PL Wadler [Jan 1992b], "The essence of functional programming," in *Proc Principles of Programming Languages*, ACM.

PL Wadler [June 1990], "Comprehending monads," in *Proc ACM Conference on Lisp and Functional Programming, Nice*, ACM.

Notes

1. Actually the distributed system is rather richer, as it includes some error handling facilities also
2. Backquotes are Haskell's notation for an infix operator.
3. 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.
4. In Haskell 1.2 the list of index-value pairs had type `[Assoc i v]`, where `Assoc` is just a pairing constructor, but Haskell 1.3 uses ordinary pairs in the list.
5. The definition of `indices` shows up an interesting shortcoming in Haskell's type signatures, which it shares with several other similar languages: there is no way to give the correct type signature to `indices`, because the type involves type variables local to the enclosing definition. We cannot write `indices :: [(i,v)]` because that would mean that `indices` has type $\forall i, v.[(i, v)]$, which is certainly not what we mean. Instead, a notation is required to allow the type variables in the signature for `newArr` to scope over the body of `newArr`. Here we simply give the type of `indices` in a comment, introduced by "`--`".
6. Notice the type signatures in comments again!
7. Indeed, the formal semantics of an imperative language often expresses the meaning of a statement as a state-to-state function.
8. Because the `let` is recursive we also have to require r to be strict
9. Indeed, we have to admit that the implementation came first!