Chapter 10

Imperative functional programming

Back in Chapter 2 we described the function `putStrLn` as being a Haskell *command*, and `IO a` as being the type of input–output *computations* that interact with the outside world and deliver values of type `a`. We also mentioned some syntax, called *`do`-notation*, for sequencing commands. This chapter explores what is really meant by these words, and introduces a new style of programming called *monadic* programming. Monadic programs provide a simple and attractive way to describe interaction with the outside world, but are also capable of much more: they provide a simple sequencing mechanism for solving a range of problems, including exception handling, destructive array updates, parsing and state-based computation. In a very real sense, a monadic style enables us to write functional programs that mimic the kind of imperative programs one finds in languages such as Python or C.

10.1 The `IO` monad

The type `IO a` is an abstract type in the sense described in the previous chapter, so we are not told how its values, which are called *actions* or commands, are represented. But you can think of this type as being

```
type IO a = World -> (a,World)
```

Thus an action is a function that takes a world and delivers a value of type `a` and a new world. The new world is then used as the input for the next action. Having changed the world with an input–output action, you can't go back to the old world. You can't duplicate the world or inspect its components. All you can do is operate on the world with given primitive actions, and put such actions together in a sequence.

One primitive action is to print a character:

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

When executed, this action prints a character on the standard output channel, usually the computer screen. For example,

```
ghci> putChar 'x'
xghci>
```

The character `x` is printed, but nothing else, so the next GHCi prompt follows without additional spaces or newlines. Performing this action produces no value of interest, so the return value is the null tuple `()`.

Another primitive action is `done :: IO ()`, which does nothing. It leaves the world unchanged and also returns the null tuple `()`.

One simple operation to sequence actions is denoted by `(>>)` and has type

```
(>>) :: IO () -> IO () -> IO ()
```

Given actions `p` and `q`, the action `p >> q` first performs action `p` and then performs action `q`. For example,

```
ghci> putChar 'x' >> putChar '\n'
x
ghci>
```

This time a newline is printed. Using `(>>)` we can define the function `putStrLn`:

```
putStrLn :: String -> IO ()
putStrLn xs = foldr (>>) done (map putChar xs) >>
              putChar '\n'
```

This action prints all the characters in a string, and then finishes up with an additional newline character. Note that `map putChar xs` is a list of actions. We are still in the universe of functional programming and its full expressive power, including uses of `map` and `foldr`, is still available to us.

Here is another primitive action:

```
getChar :: IO Char
```

When performed, this operation reads a character from the standard input channel. This channel is fed by you typing at the keyboard, so `getChar` returns the first character you type. For example,

```
ghci> getChar
x
'x'
```

After typing `getChar` and pressing return, GHCi waits for you to type a character. We typed the character `'x'` (and what we typed was echoed), and then that character was read and printed.

The generalisation of `done` is an action that does nothing and returns a named value:

```
return :: a -> IO a
```

In particular, `done = return ()`. The generalisation of `(>>)` has type

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

Given actions `p` and `q`, the action `p >> q` first does `p`, and then throws the return value away, and then does `q`. For example,

```
ghci> return 1 >> return 2
2
```

It is clear that this action is useful only when the value returned by `p` is not interesting since there is no way that `q` can depend on it. What is really wanted is a more general operator `(>>=)` with type

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```

The combination `p >>= f` is an action that, when performed, first does `p`, returning a value `x` of type `a`, then does action `f x` returning a final value `y` of type `b`. It is easy to define `(>>)` in terms of `(>>=)` and we leave this as an exercise. The operator `(>>=)` is often referred to as *bind*, though one can also pronounce it as 'then apply'.

Using `(>>=)`, we can define a function `getLine` for reading a line of input, more precisely, the list of characters up to but not including the first newline character:

```
getLine :: IO String
getLine = getChar >>= f
          where f x = if x == '\n' then return []
                      else getLine >>= g
                      where g xs = return (x:xs)
```

This has a straightforward reading: get the first character `x`; stop if `x` is a newline and return the empty list; otherwise get the rest of the line and add `x` to the front. Though the reading is straightforward, the use of nested `where` clauses makes the

definition a little clumsy. One way to make the code smoother is to use anonymous lambda expressions and instead write:

```
getLine = getChar >>= \x ->
          if x == '\n'
          then return []
          else getLine >>= \xs ->
               return (x:xs)
```

Another, arguably superior solution is to use do-notation:

```
getLine = do x <- getChar
             if x == '\n'
             then return []
             else do xs <- getLine
                     return (x:xs)
```

The right-hand side makes use of the Haskell layout convention. Note especially the indentation of the conditional expression, and the last `return` to show it is part of the inner `do`. Better in our opinion is to use braces and semicolons to control the layout explicitly:

```
getLine = do {x <- getChar;
              if x == '\n'
              then return []
              else do {xs <- getLine;
                       return (x:xs)}}
```

We return to do-notation below.

The Haskell library `System.IO` provides many more actions than just `putChar` and `getChar`, including actions to open and read files, to write and close files, to buffer output in various ways and so on. We will not go into details in this book. But perhaps two more things need to be said. Firstly, there is no function of type `IO a -> a` [1]. Once you are in a room performing input–output actions, you stay in the room and can't come out of it. To see one reason this has to be the case, suppose there is such a function, `runIO` say, and consider

```
int :: Int
int = x - y
      where x = runIO readInt
            y = runIO readInt
```

[1] Actually there is, and it's called `unsafePerformIO`, but it is a very unsafe function.

```
readInt = do {xs <- getLine; return (read xs :: Int)}
```

The action `readInt` reads a line of input and, provided the line consists entirely of digits, interprets it as an integer. Now, what is the value of `int`? The answer depends entirely on which of `x` and `y` gets evaluated first. Haskell does not prescribe whether or not `x` is evaluated before `y` in the expression `x-y`. Put it this way: input–output actions have to be sequenced in a deterministic fashion, and Haskell is a lazy functional language in which it is difficult to determine the order in which things happen. Of course, an expression such as `x-y` is a very simple example (and exactly the same undesirable phenomenon arises in imperative languages) but you can imagine all sorts of confusion that would ensue if we were provided with `runIO`.

The second thing that perhaps should be said is in response to a reader who casts a lazy eye over an expression such as

```
undefined >> return 0 :: IO Int
```

Does this code raise an error or return zero? The answer is: an error. IO is *strict* in the sense that IO actions are performed in order, even though subsequent actions may take no heed of their results.

To return to the main theme, let us summarise. The type `IO a` is an abstract type on which the following operations, at least, are available:

```
return :: a -> IO a
(>>=)  :: IO a -> (a -> IO b) -> IO b

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

The second two functions are specific to input and output, but the first two are not. Indeed they are general sequencing operations that characterise the class of types called *monads*:

```
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
```

The two monad operations are required to satisfy certain laws, which we will come to in due course. As to the reason for the name ‘monad’, it is stolen from philosophy, in particular from Leibniz, who in turn borrowed it from Greek philosophy. Don’t read anything into the name.

If that's all a monad is, then surely lots of things form a monad? Yes, indeed. In particular, the humble list type forms a monad:

```
instance Monad [] where
  return x = [x]
  xs >>= f = concat (map f xs)
```

Of course, we don't yet know what the laws governing the monad operations are, so maybe this instance isn't correct (it is), but at least the operations have the right types. Since do-notation can be used with any monad we can, for example, define the cartesian product function `cp :: [[a]] -> [[a]]` (see Section 7.3) using the new notation:

```
cp []       = return []
cp (xs:xss) = do {x <- xs;
                  ys <- cp xss;
                  return (x:ys)}
```

Comparing the right-hand side of the second clause to the list comprehension

```
[x:ys | x <- xs, ys <- cp xss]
```

one can appreciate that the two notations are very similar; the only real difference is that with do-notation the result appears at the end rather than at the beginning. If monads and do-notation had been made part of Haskell before list comprehensions, then maybe the latter wouldn't have been needed.

Here is another example. The `Maybe` type is a monad:

```
instance Monad Maybe where
  return x      = Just x
  Nothing >>= f = Nothing
  Just x >>= f  = f x
```

To appreciate what this monad can bring to the table, consider the Haskell library function

```
lookup :: Eq a => a -> [(a,b)] -> Maybe b
```

The value of `lookup x alist` is `Just y` if `(x,y)` is the first pair in `alist` with first component `x`, and `Nothing` if there is no such pair. Imagine looking up `x` in `alist`, then looking up the result `y` in a second list `blist`, and then looking up the result `z` in yet a third list `clist`. If any of these lookups return `Nothing`, then

Nothing is the final result. To define such a function we would have to write its defining expression as something like

```
case lookup x alist of
  Nothing -> Nothing
  Just y  -> case lookup y blist of
               Nothing -> Nothing
               Just z  -> lookup z clist
```

With a monad we can write

```
do {y <- lookup x alist;
    z <- lookup y blist;
    return (lookup z clist)}
```

Rather than having to write an explicit chain of computations, each of which may return Nothing, and explicitly passing Nothing back up the chain, we can write a simple monadic expression in which handling Nothing is done implicitly under a monadic hood.

do-notation

Just as list comprehensions can be translated into expressions involving map and concat, so do-expressions can be translated into expressions involving return and bind. The three main translation rules are:

```
do {p}             = p
do {p;stmts}       = p >> do {stmts}
do {x <- p;stmts} = p >>= \x -> do {stmts}
```

In these rules p denotes an action, so the first rule says that a do round a single action can be removed. In the second and third rules stmts is a *nonempty* sequence of statements, each of which is either an action or a statement of the form x <- p. The latter is *not* an action; consequently an expression such as

```
do {x <- getChar}
```

is not syntactically correct. Nor, by the way, is an empty do-expression do { }. The last statement in a do-expression must be an action.

On the other hand, the following two expressions are both fine:

```
do {putStrLn "hello "; name <- getLine; putStrLn name}
do {putStrLn "hello "; getLine; putStrLn "there"}
```

The first example prints a greeting, reads a name and completes the greeting. The second prints a greeting, reads a name but immediately forgets it, and then completes the greeting with a 'there'. A bit like being introduced to someone in real life.

Finally, there are two rules that can be proved from the translation rules above:

```
do {do {stmts}} = do {stmts}
do {stmts1; do {stmts2}} = do {stmts1; stmts2}
```

But one has to be careful; the nested `do`s in

```
do {stmts1;
    if p
    then do {stmts2}
    else do {stmts3}}
```

are necessary if `stmts2` and `stmts3` contain more than one action.

Monad laws

The monad laws say nothing much more than that expressions involving `return` and `(>>=)` simplify in just the way one would expect. There are three laws and we are going to state them in three different ways. The first law states that `return` is a right identity element of `(>>=)`:

```
(p >>= return) = p
```

In `do`-notation the law reads:

```
do {x <- p; return x} = do {p}
```

The second law says that `return` is also a kind of left identity element:

```
(return e >>= f) = f e
```

In `do`-notation the law reads:

```
do {x <- return e; f x} = do {f e}
```

The third law says that `(>>=)` is kind of associative:

```
((p >>= f) >>= g) = p >>= (\x -> (f x >>= g))
```

In `do`-notation the law reads:

```
  do {y <- do {x <- p; f x}; g y}
= do {x <- p; do {y <- f x; g y}}
= do {x <- p; y <- f x; g y}
```

The last line makes use of the un-nesting property of do-notation.

For the third way of stating the monad laws, consider the operator (>=>) defined by

```
(>=>) :: Monad m => (a -> m b) -> (b -> m c) -> (a -> m c)
(f >=> g) x = f x >>= g
```

This operator is just like function composition except that the component functions each have type x -> m y for appropriate x and y, and the order of composition is from left to right rather than from right to left. This operator, which is called *(left to right) Kleisli composition*, is defined in the Haskell library Control.Monad. There is a dual version, *(right to left) Kleisli composition*,

```
(<=<) :: Monad m => (b -> m c) -> (a -> m b) -> (a -> m c)
```

whose definition we leave as an easy exercise.

The point is that we can define (>>=) in terms of (>=>):

```
(p >>= f) = (id >=> f) p
```

More briefly, (>>=) = flip (id >=>). We also have the *leapfrog* rule:

```
(f >=> g) . h = (f . h) >=> g
```

The proof is left as an exercise.

In terms of (>=>) the three monad laws say simply that (>=>) is associative with identity return. Any set of values with an associative binary operation and an identity element is called a *monoid*, and the word 'monad' was probably adopted because of the pun with monoid. Be that as it may, this is certainly the shortest way of stating the monad laws.

One additional and instructive way of describing the monad laws is considered in the exercises.

10.3 The State monad

If it wasn't for the problem of how to sequence input–output actions correctly, monads probably wouldn't have appeared in Haskell. But once it was appreciated

what they could do, all kinds of other uses quickly followed. We have seen with the `Maybe` monad how chains of computations that involve passing information back up the chain can be simplified with monadic notation. Another primary use of monads is a way to handle *mutable* structures, such as arrays, that rely for their efficiency on being able to update their values, destroying the original structure in the process.

Mutable structures are introduced through the State-Thread monad `ST s` which we will consider in a subsequent section. Before getting on to the particular properties of this monad, we start by considering a simpler monad, called `State s`, for manipulating an explicit state `s`. You can think of the type `State s a` as being

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

An action of type `State s a` takes an initial state and returns a value of type `a` and a new state. It is tempting, but wrong, to think of `IO a` as synonymous with `State World a`. The state component `s` in `State s a` can be exposed and manipulated, but we can't expose and manipulate the world.

Specifically, as well as the monad operations `return` and `(>>=)`, five other functions are provided for working with the state monad:

```
put       :: s -> State s ()
get       :: State s s
state     :: (s -> (a,s)) -> State s a
runState  :: State s a -> (s -> (a,s))
evalState :: State s a -> s -> a
```

The function `put` puts the state into a given configuration, while `get` returns the current state. Each of these two operations can be defined in terms of `state`:

```
put s = state (\_ -> ((),s))
get   = state (\s -> (s,s))
```

On the other hand, `state` can also be defined using `put` and `get`:

```
state f = do {s <- get; let (a,s') = f s;
              put s'; return a}
```

Haskell permits an abbreviated form of `let` expressions in `do` expressions (and also in list comprehensions). We have

```
do {let decls; stmts} = let decls in do {stmts}
```

The function `runState` is the inverse of `state`: it takes both an action and an

initial state and returns the final value and the final state after performing the action (something the IO monad cannot do). The function evalState is defined by

```
evalState m s = fst (runState m s)
```

and returns just the value of the stateful computation.

Here is an example of the use of State. In Section 7.6 we constructed the following program for building a binary tree out of a given nonempty list of values:

```
build :: [a] -> BinTree a
build xs = fst (build2 (length xs) xs)
build2 1 xs = (Leaf (head xs),tail xs)
build2 n xs = (Fork u v, xs'')
              where (u,xs')  = build2 m xs
                    (v,xs'') = build2 (n-m) xs'
                    m        = n `div` 2
```

The point to appreciate here is that build2 is essentially a function that manipulates a state of type [a], returning elements of BinTree a as its result. Another way of writing build is as follows:

```
build xs = evalState (build2 (length xs)) xs

build2 :: Int -> State [a] (BinTree a)
build2 1 = do {x:xs <- get;
               put xs;
               return (Leaf x)}
build2 n = do {u <- build2 m;
               v <- build2 (n-m);
               return (Fork u v)}
           where m = n `div` 2
```

All the work in manipulating the state explicitly is done when building a leaf. The state is accessed and its first element is chosen as the label associated with a Leaf; the remaining list then is installed as the new state. Whereas the first version of build2 n threads the state explicitly, the second version hides this machinery under a monadic hood.

Notice in the first line of build2 we have a statement x:xs <- get in which the left-hand side is a *pattern* rather than a simple variable. If the current state happens to be the empty list, the action fails with a suitable error message. For example,

```
ghci>  runState (do {x:xs <- get; return x}) ""
```

```
*** Exception: Pattern match failure in do expression ...
```

Of course this behaviour cannot arise with `build2 1` because the definition only applies when the state is a singleton list. We leave it as an exercise to say what `build []` does.

As another example, consider the problem of producing a pseudo-random integer in a specified interval. Imagine we have a function

```
random :: (Int,Int) -> Seed -> (Int,Seed)
```

that takes a pair of integers as the specified interval and then a seed, and calculates a random integer and a new seed. The new seed is used for obtaining further random values. Rather than be explicit about what a seed is, suppose there is a function

```
mkSeed :: Int -> Seed
```

that makes a seed from a given integer. Now if we wanted to roll a pair of dice, we could write

```
diceRoll :: Int -> (Int,Int)
diceRoll n = (x,y)
             where (x,s1) = random (1,6) (mkSeed n)
                   (y,s2) = random (1,6) s1
```

But we could also write

```
diceRoll n = evalState (
             do {x <- randomS (1,6);
                 y <- randomS (1,6);
                 return (x,y)}
             ) (mkSeed n)
             where randomS = state . random
```

The function `randomS :: (Int,Int) -> State Seed Int` takes an interval and returns an action. The second version of `diceRoll` is a little longer than the first, but is arguably more easy to write. Imagine that instead of two dice we had five, as in liar dice. The first method would involve a chain of where-clauses expressing the linkage between five values and five seeds, something that would be easy to mistype, but the second version is easily extended and harder to get wrong.

One final point. Consider

```
evalState (do {undefined; return 0}) 1
```

Does this raise an exception, or does it return zero? In other words, is the monad

State strict, as the IO monad is, or is it lazy? The answer is that it can be both. There are two variants of the state monad, one of which is lazy and the other of which is strict. The difference lies in how the operation (>>=) is implemented. Haskell provides the lazy variant by default, in Control.Monad.State.Lazy, but you can ask for the strict variant, in Control.Monad.State.Strict if you want.

10.4 The ST monad

The state-thread monad, which resides in the library Control.Monad.ST, is a different kettle of fish entirely from the state monad, although the kettle itself looks rather similar. Like State s a you can think of this monad as the type

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

but with one very important difference: the type variable s cannot be instantiated to specific states, such as Seed or [Int]. Instead it is there only to *name* the state. Think of s as a label that identifies one particular state *thread*. All mutable types are tagged with this thread, so that actions can only affect mutable values in their own state thread.

One kind of mutable value is a *program variable*. Unlike variables in Haskell, or mathematics for that matter, program variables in imperative languages can change their values. They can be thought of as *references* to other values, and in Haskell they are entities of type STRef s a. The s means that the reference is local to the state thread s (and no other), and the a is the type of value being referenced. There are operations, defined in Data.STRef, to create, read from and write to references:

```
newSTRef   :: a -> ST s (STRef s a)
readSTRef  :: STRef s a -> ST s a
writeSTRef :: STRef s a -> a -> ST s ()
```

Here is an example. Recall Section 7.6 where we gave the following definition of the Fibonacci function:

```
fib :: Int -> Integer
fib n  = fst (fib2 n)
fib2 0 = (0,1)
fib2 n = (b,a+b)  where (a,b) = fib2 (n-1)
```

Evaluating `fib` takes linear time, but the space involved is not constant (even ignoring the fact that arbitrarily large integers cannot be stored in constant space): each recursive call involves fresh variables `a` and `b`. By contrast, here is a definition of `fib` in the imperative language Python:

```
def fib (n):
    a,b = 0,1
    for i in range (0,n):
        a,b = b,a+b
    return a
```

The definition manipulates two program variables `a` and `b`, and runs in constant space (at least, for small integers). We can translate the Python code almost directly into Haskell:

```
fibST :: Int -> ST s Integer
fibST n = do {a <- newSTRef 0;
              b <- newSTRef 1;
              repeatFor n
               (do {x <- readSTRef a;
                    y <- readSTRef b;
                    writeSTRef a y;
                    writeSTRef b $! (x+y)});
              readSTRef a}
```

Note the use of the strict application operator `($!)` to force evaluation of the sum. The action `repeatFor` repeats an action a given number of times:

```
repeatFor :: Monad m => Int -> m a -> m ()
repeatFor n = foldr (>>) done . replicate n
```

All well and good, but we end up with an action `ST s Integer` when what we really want is an integer. How do we escape from the monad back into the world of Haskell values?

The answer is to provide a function similar to `runState` for the state monad, Here it is, with its type:

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

This type is unlike any other Haskell type we have met so far. It is what is called a *rank 2 polymorphic type*, while all previous polymorphic types have had rank 1. What it says is that the argument of `runST` must be universal in `s`, so it can't

depend on any information about s apart from its name. In particular, every STRef declared in the action has to carry the same thread name s.

To amplify a little on rank 2 types, consider the difference between the two lists

```
list1 :: forall a. [a -> a]
list2 :: [forall a. a -> a]
```

The type of list1 is just what we would have previously written as [a -> a] because in ordinary rank 1 types universal quantification at the outermost level is assumed. For example, [sin,cos,tan] is a possible value of list1 with the instantiation Float for a. But there are only two functions that can be elements of list2, namely id and the undefined function undefined, because these are the only two functions with type forall a. a -> a. If you give me an element x of a type a about which absolutely nothing is known, the only things I can do if I have to give you back an element of a, is either to give you x or $\perp$.

Why have a rank 2 type for runST? Well, it prevents us from defining things like

```
let v = runST (newSTRef True)
in runST (readSTRef v)
```

This code is not well-typed because

```
newSTRef True :: ST s (STref s Bool)
```

and in the expression runST (newSTRef Bool) the Haskell type checker cannot match STRef s a with a, the expected result type of runST. Values of type STRef s a cannot be exported from ST s, but only entities whose types do not depend on s. If the code were allowed, then the reference allocated in the first runST would be usable inside the second runST. That would enable reads in one thread to be used in another, and hence the result would depend on the evaluation order used to execute the threads, leading to mayhem and confusion. It is just the same problem that we prevented from occurring in the IO monad.

But we can safely define

```
fib :: Int -> Integer
fib n = runST (fibST n)
```

This version of fib runs in constant space.

For our purposes the main use of the ST monad resides in its ability to handle mutable arrays. The whole question of arrays deserves a section to itself.

10.5 Mutable arrays

It sometimes surprises imperative programmers who meet functional programming for the first time that the emphasis is on lists as the fundamental data structure rather than arrays. The reason is that most uses of arrays (though not all) depend for their efficiency on the fact that updates are destructive. Once you update the value of an array at a particular index the old array is lost. But in functional programming, data structures are *persistent* and any named structure continues to exist. For instance, `insert x t` may insert a new element `x` into a tree `t`, but `t` continues to refer to the original tree, so it had better not be overwritten.

In Haskell a mutable array is an entity of type `STArray s i e`. The `s` names the state thread, `i` the index type and `e` the element type. Not every type can be an index; legitimate indices are members of the type class `Ix`. Instances of this class include `Int` and `Char`, things that can be mapped into a contiguous range of integers.

Like `STRefs` there are operations to create, read from and write to arrays. Without more ado we consider an example, explaining the actions as we go along. Recall the Quicksort algorithm from Section 7.7:

```
qsort :: (Ord a) => [a] -> [a]
qsort []     = []
qsort (x:xs) = qsort [y | y <- xs, y < x] ++ [x] ++
               qsort [y | y <- xs, x <= y]
```

There we said that when Quicksort is implemented in terms of arrays rather than lists, the partitioning phase can be performed *in place* without using any additional space. We now have the tools to write just such an algorithm. We begin with

```
qsort :: (Ord a) => [a] -> [a]
qsort xs = runST $
           do {xa <- newListArray (0,n-1) xs;
               qsortST xa (0,n);
               getElems xa}
           where n = length xs
```

First we create a mutable array with bounds `(0,n-1)` and fill it with the elements of `xs`. Sorting the array is done with the action `qsortST xa (0,n)`. At the end, the list of elements of the sorted array is returned. In the code above, the action `newListArray` has type

```
Ix i => (i, i) -> [e] -> ST s (STArray s i e)
```

and `getElems` has type

```
Ix i => STArray s i e -> ST s [e]
```

The first constructs a mutable array from a list of elements, and the second returns a list of the elements in a mutable array.

The purpose of `qsortST xa (a,b)` is to sort the elements in the sub-array of `xa` in the interval `(a,b)`, where by definition such an interval includes the lower bound but excludes the upper bound; in other words `[a .. b-1]`. Choosing intervals that are closed on the left but open on the right is almost always the best policy when processing arrays. Here is the definition of `qsortST`:

```
qsortST :: Ord a => STArray s Int a ->
           (Int,Int) -> ST s ()
qsortST xa (a,b)
  | a == b    = return ()
  | otherwise = do {m <- partition xa (a,b);
                    qsortST xa (a,m);
                    qsortST xa (m+1,b)}
```

If `a==b` we have an empty interval and there is nothing to do. Otherwise we rearrange the array so that for some suitable element `x` in the array all elements in the interval `(a,m)` are less than `x`, and all elements in the interval `(m+1,b)` are at least `x`. The element `x` itself is placed in the array at position `m`. Sorting is then completed by sorting both sub-intervals.

It remains to define `partition`. The *only* way to find a suitable definition is by formal development using pre- and post-conditions and loop invariants. But this is a book on functional programming, not on the formal development of imperative programs, so we are going to cop out and just record one version:

```
partition xa (a,b)
 = do {x <- readArray xa a;
       let loop (j,k)
            = if j==k
              then do {swap xa a (k-1);
                       return (k-1)}
              else do {y <- readArray xa j;
                       if y < x then loop (j+1,k)
                       else do {swap xa j (k-1);
                                loop (j,k-1)}}
       in loop (a+1,b)}
```

The action `swap` is defined by

```
swap :: STArray s Int a -> Int -> Int -> ST s ()
swap xa i j =  do {v <- readArray xa i;
                   w <- readArray xa j;
                   writeArray xa i w;
                   writeArray xa j v}
```

Here is a brief and certainly inadequate explanation of how `partition` works. We begin by taking the first element `x` in the interval `(a,b)` as pivot. We then enter a loop that processes the remaining interval `(a+1,b)`, stopping when the interval becomes empty. We pass over elements that are less than `x`, shrinking the interval from the left. Encountering a `y` not less than `x`, we swap it with the element at the rightmost position in the interval, shrinking the interval from the right. When the interval becomes empty, we place the pivot in its final position, returning that position as a result.

Note that `loop` is defined as a local procedure within the monad. We could have defined it as a global procedure, though we would have had to add three extra parameters, namely the array `xa`, the pivot `x` and the starting position `a`.

Hash tables

A purely functional Quicksort has the same asymptotic time efficiency as one based on mutable arrays, but there are one or two places where mutable arrays seem to play a crucial role in achieving an asymptotically faster algorithm. One such place is the use of hash tables for an efficient representation of sets.

But let us approach the use of hash tables in the context of a particular problem. Consider a typical puzzle defined in terms of two finite sets, a set of *positions* and a set of *moves*. Given are the following functions:

```
moves  :: Position -> [Move]
move   :: Position -> Move -> Position
solved :: Position -> Bool
```

The function `moves` describes the set of possible moves that can be made in a given position, `move` makes a move, and `solved` determines those positions that are a solution to the puzzle. Solving the puzzle means finding some sequence of moves, preferably a shortest such sequence, that leads from a given starting position to a solved position:

```
solve :: Position -> Maybe [Move]
```

The value `solve p` is `Nothing` if there is no sequence of moves starting in position `p` that leads to a solved position, and `Just ms` otherwise, where

```
solved (foldl move p ms)
```

We are going to implement `solve` by carrying out a *breadth-first* search. What this means is that we examine all positions one move away from the starting position to see if there is a solution, then all positions two moves away, and so on. Breadth-first will therefore find a shortest solution if one exists. To implement the search we need

```
type Path     = ([Move],Position)
type Frontier = [Path]
```

A path consists of a sequence of moves made from the starting position (in reverse order), and the position that results after making the moves. A frontier is a list of paths waiting to be extended into longer paths. A breadth-first search is then implemented by

```
solve p = bfs [] [([],p)]

bfs :: [Position] -> Frontier -> Maybe [Move]
bfs ps [] = Nothing
bfs ps ((ms,p):mps)
 | solved p    = Just (reverse ms)
 | p `elem` ps = bfs ps mps
 | otherwise   = bfs (p:ps) (mps ++ succs (ms,p))

succs :: Path -> [Path]
succs (ms,p) = [(m:ms,move p m) | m <- moves p]
```

The first argument `ps` of `bfs` represents the set of positions that have already been explored. The second argument is the frontier, which is managed in a queue-like fashion to ensure that paths of the same length are inspected before their successors. Inspecting a path means accepting it if the final position is a solution, rejecting it if the end position has already been explored, and otherwise adding its successors to the end of the current frontier for future exploration. The moves in a successful path are reversed before being returned as the final result of `bfs` simply because, for efficiency, `succs` adds a new move to the front of the list rather than at the end.

There are two major sources of inefficiency with `bfs`, one concerning the use of

(++) and the other concerning elem. Firstly, the size of a frontier can grow exponentially and so concatenating successors to the end of the frontier is slow. Better is the following alternative to bfs:

```
bfs :: [Position] -> Frontier -> Frontier ->
       Maybe [Move]
bfs ps [] []  = Nothing
bfs ps [] mqs = bfs ps mqs []
bfs ps ((ms,p):mps) mqs
 | solved p    = Just (reverse ms)
 | p `elem` ps = bfs ps mps mqs
 | otherwise   = bfs (p:ps) mps (succs (ms,p) ++ mqs)
```

The additional argument is a temporary frontier used to store successors. When the first frontier is exhausted the contents of the temporary frontier are installed as the new frontier. Adding successors to the front of the temporary frontier takes time proportional to the number of successors, not to the size of the frontier, and that leads to a faster algorithm. On the other hand, the new version of bfs is not the same as the old one because successive frontiers are traversed alternately from left to right and from right to left. Nevertheless a shortest solution will still be found if one exists.

The second source of inefficiency is the membership test. Use of a list to store previously explored positions is slow because the membership test can take time proportional to the number of currently explored positions. It would all be easier if positions were integers in the range $[0\,..\,n-1]$ for some n, for then we could use a boolean array with bounds $(0,n-1)$ to tick off positions as they arise. The membership test would then consist of a single array lookup.

One can imagine coding positions as integers, but not as integers in an initial segment of the natural numbers. For instance, a Sudoku position (see Chapter 5) can be expressed as an integer consisting of 81 digits. So suppose we have a function

```
encode :: Position -> Integer
```

that encodes positions as integers. To reduce the range we can define

```
hash :: Position -> Int
hash p = fromInteger (encode p) `mod` n
```

for some suitable n :: Int. The result of hash is then an integer in the range [0..n-1].

The one hitch, and it's a big one, is that two distinct positions may hash to the

same integer. To solve this problem we abandon the idea of having an array of booleans, and instead have an array of lists of positions. The positions in the array at index k are all those whose hash value is k. There is no guarantee that any of this will improve efficiency in the worst case, but if we allow n to be reasonably large, and trust that the hash function assigns integers to positions in a reasonably evenly distributed way, then the complexity of a membership test is reduced by a factor of n.

With this hashing scheme the revised code for `solve` is:

```
solve :: Maybe [Move]
solve = runST $
        do {pa <- newArray (0,n-1) [];
            bfs pa [([],start)] []}

bfs :: STArray s Int [Position] -> Frontier ->
       Frontier -> ST s (Maybe [Move])
bfs pa [] []  = return Nothing
bfs pa [] mqs = bfs pa mqs []
bfs pa ((ms,p):mps) mqs
 = if solved p then return (Just (reverse ms))
   else do {ps <- readArray pa k;
            if p `elem` ps
            then bfs pa mps mqs
            else
            do {writeArray pa k (p:ps);
                bfs pa mps (succs (ms,p) ++ mqs)}}
   where k = hash p
```

10.6 Immutable arrays

We cannot leave the subject of arrays without mentioning a very nice Haskell library `Data.Array` that provides purely functional operations on immutable arrays. The operations are implemented using mutable arrays, but the interface is purely functional.

The type `Array i e` is an abstract type of arrays with indices of type `i` and elements of type `e`. One basic operation for constructing arrays is

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

This function take a pair of bounds, the lowest and highest indices in the array, and a list of index-element pairs specifying the array entries. The result is an array with the given bounds and entries. Any entry missing from the association list is deemed to be the undefined entry. If two entries have the same index, or one of the indices is out of bounds, the undefined array is returned. Because of these checks, array construction is strict in the indices, though lazy in the elements. Building the array takes linear time in the number of entries.

A simple variant of `array` is `listArray` which takes just a list of elements:

```
listArray :: Ix i => (i,i) -> [e] -> Array i e
listArray (l,r) xs = array (l,r) (zip [l..r] xs)
```

Finally, there is another way of building arrays called `accumArray` whose type appears rather daunting:

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

The first argument is an 'accumulating' function for transforming array entries and new values into new entries. The second argument is an initial entry for each index. The third argument is a pair of bounds, and the fourth and final argument is an association list of index–value pairs. The result is an array built by processing the association list from left to right, combining entries and values into new entries using the accumulating function. The process takes linear time in the length of the association list, assuming the accumulating function takes constant time.

That's what `accumArray` does in words. In symbols,

```
elems (accumArray f e (l,r) ivs)
  = [foldl f e [v | (i,v) <- ivs, i==j] | j <- [l..r]]
```

where `elems` returns the list of elements of an array in index order. Well, the identity above is not quite true: there is an additional restriction on `ivs`, namely that every index should lie in the specified range. If this condition is not met, then the left-hand side returns an error while the right-hand side does not.

Complicated as `accumArray` seems, it turns out to be a very useful tool for solving certain kinds of problem. Here are two examples. First, consider the problem of representing directed graphs. Directed graphs are usually described in mathematics in terms of a set of *vertices* and a set of *edges*. An edge is an ordered pair (j,k) of vertices signifying that the edge is directed from j to k. We say that k is *adjacent* to j. We will suppose that vertices are named by integers in the range 1 to n for some n. Thus

```
type Vertex = Int
```

```
type Edge   = (Vertex,Vertex)
type Graph  = ([Vertex],[Edge])

vertices g = fst g
edges g    = snd g
```

In computing, directed graphs are often described in terms of adjacency lists:

```
adjs :: Graph -> Vertex -> [Vertex]
adjs g v = [k | (j,k) <- edges g, j==v]
```

The problem with this definition of `adjs` is that it takes time proportional to the number of edges to compute the adjacency list of any particular vertex. Better is to implement `adjs` as an array:

```
adjArray :: Graph -> Array Vertex [Vertex]
```

Then we have

```
adjs g v = (adjArray g)!v
```

where `(!)` denotes the operation of array-indexing. For reasonably sized arrays this operation takes constant time.

The specification of `adjArray` is that

```
elems (adjArray g)
 = [[k | (j,k) <- edges g, j==v] | v <- vertices g]
```

Using this specification we can calculate a direct definition of `adjArray`. To keep each line short, abbreviate `edges g` to `es` and `vertices g` to `vs`, so

```
elems (adjArray g) = [[k | (j,k) <- es, j==v] | v <- vs]
```

Concentrating on the right-hand side, the first step is to rewrite it using the law `foldr (:) [] = id`. That gives the expression

```
[foldr (:) [] [k | (j,k) <- es, j==v] | v <- vs]
```

Next we use the law `foldr f e xs = foldl (flip f) e (reverse xs)` for all finite lists `xs`. Abbreviating `flip (:)` to `(@)`, we obtain

```
[foldl (@) [] (reverse [k | (j,k) <- es, j==v]) | v <- vs]
```

Distributing `reverse` we obtain the expression

```
[foldl (@) [] [k | (j,k) <- reverse es, j==v] | v <- vs]
```

Next we use `swap (j,k) = (k,j)` to obtain

```
[foldl (@) [] [j | (k,j) <- es', j==v] | v <- vs]
```

where `es' = map swap (reverse es)`. Finally, using `n = length vs` and the specification of `accumArray`, we obtain

```
  elems (adjArray g)
= elems (accumArray (flip (:)) [] (1,n) es')
```

That means we can define

```
adjArray g = accumArray (flip (:)) [] (1,n) es
             where n  = length (vertices g)
                   es = map swap (reverse (edges g))
```

This definition of `adjArray g` computes the successors in time proportional to the number of edges.

Here is the second example of the use of `accumArray`. Suppose we are given a list of n integers, all in the range $(0,m)$ for some m. We can sort this list in $\Theta(m+n)$ steps by counting the number of times each element occurs:

```
count :: [Int] -> Array Int Int
count xs = accumArray (+) 0 (0,m) (zip xs (repeat 1))
```

The value `repeat 1` is an infinite list of `1`s. Counting takes $\Theta(n)$ steps. Having counted the elements, we can now sort them:

```
sort xs = concat [replicate c x
                 | (x,c) <- assocs (count xa)]
```

The function `assocs` is yet another library function and returns the list of index–element pairs of an array in index order. The sorting is completed in $\Theta(m)$ steps.

As well as the above operations `Data.Array` contains one or two more, including the update operation `(//)`:

```
(//) :: Ix i => Array i e -> [(i,e)] -> Array i e
```

For example, if `xa` is an $n \times n$ matrix, then

```
xa // [((i,i),0) | i <- [1..n]]
```

is the same matrix except with zeros along the diagonal. The downside of `(//)` is that it takes time proportional to the size of the array, even for an update involving a single element. The reason is that a completely new array has to be constructed because the old array `xa` continues to exist.

We have ended the chapter back in the world of pure functional programming,

where equational reasoning can be used both to calculate definitions and to optimise them. Although the monadic style is attractive to programmers who are used to imperative programming, there remains the problem of how to reason about monadic programs. True, equational reasoning is still possible in certain situtations (see Exercise F for an example), but it is not so widely applicable as it is in the pure functional world (witness the correctness of the partition phase of Quicksort). Imperative programmers have the same problem, which they solve (if they bother to) by using predicate calculus, preconditions, postconditions and loop invariants. How to reason directly with monadic code is still a topic of ongoing research.

Our best advice is to use the monadic style sparingly and only when it is really useful; otherwise the most important aspect of functional programming, the ability to reason mathematically about its constructs, is lost.

10.7 Exercises

Exercise A

Recall that

```
putStr = foldr (>>) done . map putChar
```

What does

```
foldl (>>) done . map putChar
```

do? Justify your answer by expressing `(>>)` in terms of `(>>=)` and appealing to the monad laws.

Exercise B

Using a pattern-matching style, define a function

```
add3 :: Maybe Int -> Maybe Int -> Maybe Int -> Maybe Int
```

that adds three numbers, provided all of them exist. Now rewrite `add3` using the `Maybe` monad.

Exercise C

The monadic definition of `cp` in Section 10.1 is still inefficient. We might prefer to write

```
cp (xs:xss) = do {ys <- cp xss;
                  x <- xs;
                  return (x:ys)}
```

By definition a *commutative* monad is one in which the equation

```
   do {x <- p; y <- q; f x y}
 = do {y <- q; x <- p; f x y}
```

holds. The `IO` monad is certainly not commutative, while some other monads are. Is the `Maybe` monad commutative?

Exercise D

Every monad is a functor. Complete the definition

```
instance Monad m => Functor m where
  fmap :: (a -> b) -> m a -> m b
  fmap f = ...
```

Currently Haskell does not insist that the `Monad` class should be a subclass of `Functor`, though there are plans to change this in future releases. Instead, Haskell provides a function `liftM` equivalent to `fmap` for monads. Give a definition of `liftM` in terms of `return` and `>>=`.

The function `join :: m (m a) -> m a` flattens two layers of monadic structure into one. Define `join` in terms of `>>=`. What familiar functions do `join` and `liftM` give for the list monad?

Finally, using `join` and `liftM`, define `(>>=)`. It follows that instead of defining monads in terms of `return` and `>>=`, we can also define them in terms of `return`, `liftM` and `join`.

Exercise E

A number of useful monadic functions are provided in the `Control.Monad` library. For instance:

```
sequence_ :: Monad m => [m a] -> m ()
sequence_ = foldr (>>)  done
```

(The underscore convention is used in a number of places in Haskell to signify that the result of the action is the null tuple.) Define the related function

```
sequence :: Monad m => [m a] -> m [a]
```

Using these two functions, define

```
mapM_ :: Monad m => (a -> m b) -> [a] -> m ()
mapM  :: Monad m => (a -> m b) -> [a] -> m [b]
```

Also, define

```
foldM :: Monad m => (b -> a -> m b) -> b -> [a] -> m b
```

In the text we made use of a function `repeatFor n` that repeated an action `n` times. Generalise this function to

```
for_ :: Monad m => [a] -> (a -> m b) -> m ()
```

Exercise F

Here is an exercise in monadic equational reasoning. Consider the function

```
add :: Int -> State Int ()
add n = do {m <- get; put (m+n)}
```

The task is to prove that

```
sequence_ . map add = add . sum
```

where `sequence_` was defined in the previous exercise and `sum` sums a list of integers. You will need the fusion law of `foldr`, some simple laws of `put` and `get`, and the monad law

```
do {stmts1} >> do {stmts2} = do {stmts1;stmts2}
```

which is valid provided the variables in `stmts1` and `stmts2` are disjoint.

Exercise G

Prove the leapfrog rule: `(f >=> g) . h = (f . h) >=> g`.

Using this rule, prove: `(return . h) >=> g = g . h`.

Exercise H

Prove that

```
liftM f = id >=> (return . f)
join    = id >=> id
```

A fourth way of describing the monad laws is in terms of the two functions `liftM` and `join` of Exercise D. There are seven laws governing these two functions, all of which have a familiar ring:

```
liftM id      = id
liftM (f . g) = liftM f . liftM g

liftM f . return = return . f
liftM f . join   = join . liftM (liftM f)

join . return       = id
join . liftM return = id
join . liftM join   = join . join
```

Prove the fourth rule.

Exercise I

What does `build []` do (see Section 10.3)?

Exercise J

Write an interactive program to play hangman. An example session:

```
ghci> hangman
I am thinking of a word:
-----
Try and guess it.
guess: break
-a---
guess: parties
Wrong number of letters!
guess: party
-appy
guess: happy
You got it!
Play again? (yes or no)
no
Bye!
```

Assume that a list of secret words is stored in a file called `Words`, so that the action `xs <- readFile "Words"` reads the file as a list of characters. By the way, `readFile` is lazy in that its contents are read on demand.

Exercise K

Write another version of `fib` in terms of a `fibST` that uses a single `STRef`.

Exercise L

One way of defining the greatest common divisor (`gcd`) of two positive integers is:

```
gcd (x,y) | x==y = x
          | x<y  = gcd (x,y-x)
          | x>y  = gcd (x-y,y)
```

Translate this definition into two other programs, one of which uses the `State` monad and the other the `ST` monad.

Exercise M

Here is a concrete puzzle you can solve using breadth-first search. A cut-down version of Sam Loyd's famous 15 puzzle is the 8 puzzle. You are given a 3×3 array containing tiles numbered from 1 to 8 and one blank space. You move by sliding an adjacent tile into the blank space. Depending on where the blank space is, you can slide tiles upwards, downwards, to the left or to the right. At the start the blank space is in the top left corner and the tiles read from 1 to 8. At the end the blank space is in the bottom right corner, but the tiles are still neatly arranged in the order 1 to 8.

Your mission, should you choose to accept it, is to settle on a suitable representation of positions and moves, and to define the functions `moves`, `move`, `solved` and `encode`.

10.8 Answers

Answer to Exercise A

We claim that `(>>) :: IO () -> IO () -> IO ()` is associative with identity element `done`. That means

```
putStr xs = foldl (>>) done (map putChar xs)
```

for all finite strings `xs`

We concentrate on the proof of associativity. Firstly, for actions in `IO ()` we have

```
p >> q = p >>= const q
```

where `const x y = x`. Now we can reason:

```
  (p >> q) >> r
=   {definition of (>>)}
  (p >>= const q) >>= const r
=   {third monad law}
  p >>= const (q >>= const r)
=   {definition of (>>)}
  p >>= const (q >> r)
=   {definition of (>>)}
  p >> (q >> r)
```

Answer to Exercise B

The direct version uses pattern matching with a wild-card:

```
add3 Nothing _  _                = Nothing
add3 (Just x) Nothing _          = Nothing
add3 (Just x) (Just y) Nothing   = Nothing
add3 (Just x) (Just y) (Just z) = Just (x+y+z)
```

This definition ensures that `add Nothing undefined = Nothing`.

The monadic version reads:

```
add3 mx my mz
 = do {x <- mx; y <- my; z <- mz;
       return (x + y + z)}
```

Answer to Exercise C

Yes. The commutative law states that

```
   p >>= \x -> q >>= \y -> f x y
 = q >>= \y -> p >>= \x -> f x y
```

In the `Maybe` monad there are four possible cases to check. For example, both sides simplify to `Nothing` if `p = Nothing` and `q = Just y`, . The other cases are similar.

Answer to Exercise D

We have

```
fmap f p = p >>= (return . f)
join p   = p >>= id
```

For the list monad we have `liftM = map` and `join = concat`.

In the other direction

```
p >>= f = join (liftM f p)
```

Answer to Exercise E

The function `sequence` is defined by

```
sequence :: Monad m => [m a] -> m [a]
sequence = foldr k (return [])
 where k p q = do {x <- p; xs <- q; return (x:xs)}
```

The two new map functions are:

```
mapM_ f = sequence_ . map f
mapM f  = sequence . map f
```

The function `foldM` is defined by

```
foldM :: Monad m => (b -> a -> m b) ->
          b -> [a] -> m b
foldM f e []     = return e
foldM f e (x:xs) = do {y <- f e x; foldM f y xs}
```

Note that `foldM` is analogous to `foldl` in that it works from left to right. Finally `for = flip mapM_`.

Answer to Exercise F

The first thing to note is that

```
  sequence_ . map add
= foldr (>>) done . map add
= foldr ((>>) . add) done
```

using the fusion law of `foldr` and `map` given in Section 6.3. Moreover,

```
((>>) . add) n p = add n >> p
```

Since `sum = foldr (+) 0` that means we have to prove

```
foldr (\ n p -> add n >> p) = add . foldr (+) 0
```

That looks like an instance of the fusion law of `foldr`. We therefore have to show that `add` is strict (which it is), and

```
add 0 = done
add (n + n') = add n >> add n'
```

Here goes:

```
  add 0
=   {definition}
  do {m <- get; put (m+0)}
=   {arithmetic}
  do {m <- get; put m}
=   {simple law of put and get}
  done
```

That disposes of the first condition. For the second we start with the more complicated side and reason:

```
  add n >> add n'
=   {definition}
  do {l <- get; put (l + n) } >>
  do {m <- get; put (m + n')}
=   {monad law}
  do {l <- get; put (l + n); m <- get; put (m + n')}
=   {simple law of put and get}
  do {l <- get; put ((l + n) + n')}
=   {associativity of (+); definition of add}
  add (n + n')
```

Answer to Exercise G

We can reason:

```
  (f >=> g) (h x)
=   {definition of (>=>)}
  f (h x) >>= g
=   {definition of (>=>)}
  (f . h >=> g) x
```

For the second part:

```
  (return . h) >=> g
=   {leapfrog rule}
  (return >=> g) . h
=   {monad law}
  g . h
```

Answer to Exercise H

For the fourth rule we simplify both sides. For the left-hand side:

```
  liftM f . join
=   {definitions}
  (id >=> (return . f)) . (id >=> id)
=   {leapfrog rule and id . f = f}
  (id >=> id) >=> (return . f)
```

For the right-hand side:

```
  join . liftM (liftM f)
=   {definitions}
  (id >=> id) . (id >=> return . (id >=> (return . f)))
=   {leapfrog rule, and associativity of (>=>)}
  id >=> (return . (id >=> (return . f))) >=> id
=   {since (return . h) >=> g = g . h}
  id >=> id >=> (return . f)
```

The two sides are equal because `(>=>)` is associative.

Answer to Exercise I

`build []` causes an infinite loop, so its value is $\perp$.

Answer to Exercise J

For the main function we can define

```
hangman :: IO ()
hangman = do {xs <- readFile "Words";
              play (words xs)}
```

The function play plays as many rounds of the game as desired with different words from the file (which we quietly suppose always has enough words):

```
play (w:ws)
= do {putStrLn "I am thinking of a word:";
      putStrLn (replicate (length w) '-');
      putStrLn "Try and guess it.";
      guess w ws}
```

The function guess deals with a single guess, but keeps the remaining words for any subsequent round of play:

```
guess w ws
= do {putStr "guess: ";
      w' <- getLine;
      if length w' /= length w then
      do {putStrLn "Wrong number of letters!";
          guess w ws}
      else if w' == w
      then
      do {putStrLn "You got it!";
          putStrLn "Play again? (yes or no)";
          ans <- getLine;
          if ans == "yes"
          then play ws
          else putStrLn "Bye!"}
      else do {putStrLn (match w' w);
               guess w ws}}
```

Finally we program match:

```
match w' w = map check w
 where
 check x = if x `elem` w' then x else '-'
```

Answer to Exercise K

The following program is correct but doesn't run in constant space:

```
fib n = fst $ runST (fibST n)

fibST :: Int -> ST s (Integer,Integer)
fibST n = do {ab <- newSTRef (0,1);
```

```
repeatFor n
 (do {(a,b) <- readSTRef ab;
      writeSTRef ab $! (b,a+b)});
readSTRef ab}
```

The reason is that `(b,a+b)` is already in head-normal form, so strict-apply has no effect. The penultimate line needs to be changed to

```
b `seq` (a+b) `seq` writeSTRef ab (b,a+b)
```

in order to force evaluation of the components.

Answer to Exercise L

The version that uses the `State` monad:

```
gcd (x,y) = fst $ runState loop (x,y)

loop :: State (Int,Int) Int
loop = do {(x,y) <- get;
           if x == y
           then return x
           else if x < y
           then do {put (x,y-x); loop}
           else do {put (x-y,y); loop}}
```

The version that uses the `ST` monad:

```
gcd (x,y) = runST $
            do {a <- newSTRef x;
                b <- newSTRef y;
                loop a b}

loop :: STRef s Int -> STRef s Int -> ST s Int
loop a b
 = do {x <- readSTRef a;
       y <- readSTRef b;
       if x==y
       then return x
       else if x<y
       then do {writeSTRef b (y-x);loop a b}
       else do {writeSTRef a (x-y);loop a b}}
```

Answer to Exercise M

There are, of course, many possible answers. The one I chose was to represent the array of tiles by a list of nine digits $[0\,..\,8]$ with zero representing the space. To avoid recalculation, a position is represented by a pair (j, ks) with j as the position of the zero in ks, where ks was some permutation of $[0\,..\,8]$. Thus:

```
type Position = (Int,[Int])
data Move     = Up | Down | Left | Right

encode :: Position -> Integer
encode (j,ks) = foldl op 0 ks
  where op x d = 10*x + fromIntegral d

start :: Position
start = (0,[0..8])
```

The function `moves` can be defined by

```
moves :: Position -> [Move]
moves (j,ks)
  = [Up    | j `notElem` [6,7,8]] ++
    [Down  | j `notElem` [0,1,2]] ++
    [Left  | j `notElem` [2,5,8]] ++
    [Right | j `notElem` [0,3,6]]
```

Up moves are allowed except for a blank in the bottom row; down moves except for a blank in the top row, left moves except for a blank in the rightmost column, and right moves except for a blank in the leftmost column.

The function `move` can be defined by:

```
move :: Position -> Move -> Position
move (j,ks) Up    = (j+3,swap (j,j+3) ks)
move (j,ks) Down  = (j-3,swap (j-3,j) ks)
move (j,ks) Left  = (j+1,swap (j,j+1) ks)
move (j,ks) Right = (j-1,swap (j-1,j) ks)

swap (j,k) ks = ks1 ++ y:ks3 ++ x:ks4
 where (ks1,x:ks2) = splitAt j ks
       (ks3,y:ks4) = splitAt (k-j-1) ks2
```

Finally,

```
solved :: Position -> Bool
solved p = p == (8,[1,2,3,4,5,6,7,8,0])
```

My computer produced:

```
ghci> solve start
Just [Left,Up,Right,Up,Left,Left,Down,
      Right,Right,Up,Left,Down,Down,Left,
      Up,Up,Right,Right,Down,Left,Left,Up]
(4.84 secs, 599740496 bytes)
```

10.9 Chapter notes

Read *The History of Haskell* to see how monads came to be an integral part of Haskell, and why this idea has been mainly responsible for the increasing use of Haskell in the real world. Monads are used to structure GHC, which itself is written in Haskell. Each phase of the compiler uses a monad for book-keeping information. For instance, the type checker uses a monad that combines state (to maintain a current substitution), a name supply (for fresh type variable names) and exceptions.

Use of `do`-notation in preference to `(>>=)` was suggested by John Launchbury in 1993 and was first implemented by Mark Jones in Gofer.

The number of tutorials on monads has increased steadily over the years; see

```
haskell.org/haskellwiki/Monad_tutorials
```

for a reasonably comprehensive list.

The example (in Exercise F) of monadic equational reasoning can be found in the paper 'Unifying theories of programming with monads', (UTP Symposium, August 2012) by Jeremy Gibbons. For additional material on reasoning equationally with monads, read 'Just `do` it: simple monadic equational reasoning' by Jeremy Gibbons and Ralf Hinze, which appeared in the proceedings of the 2011 International Conference of Functional Programming. Both papers can be found at

```
www.cs.ox.ac.uk/people/jeremy.gibbons/publications/
```