IO inside

From HaskellWiki

Haskell I/O has always been a source of confusion and surprises for new Haskellers. While simple I/O code in Haskell looks very similar to its equivalents in imperative languages, attempts to write somewhat more complex code often result in a total mess. This is because Haskell I/O is really very different internally. Haskell is a pure language and even the I/O system can't break this purity.

The following text is an attempt to explain the details of Haskell I/O implementations. This explanation should help you eventually master all the smart I/O tricks. Moreover, I've added a detailed explanation of various traps you might encounter along the way. After reading this text, you will receive a "Master of Haskell I/O" degree that is equal to a Bachelor in Computer Science and Mathematics, simultaneously.

If you are new to Haskell I/O you may prefer to start by reading the Introduction to IO page.

Contents

1 Haskell is a pure language

Haskell is a pure language, which means that the result of any function call is fully determined by its arguments. Pseudo-functions like rand() or getchar() in C, which return different results on each call, are simply impossible to write in Haskell. Moreover, Haskell functions can't have side effects, which means that they can't effect any changes to the "real world", like changing files, writing to the screen, printing, sending data over the network, and so on. These two restrictions together mean that any function call can be replaced by the result of a previous call with the same parameters, and the language **guarantees** that all these rearrangements will not change the program result!

Let's compare this to C: optimizing C compilers try to guess which functions have no side effects and don't depend on mutable global variables. If this guess is wrong, an optimization can change the program's semantics! To avoid this kind of disaster, C optimizers are conservative in their guesses or require hints from the programmer about the purity of functions.

Compared to an optimizing C compiler, a Haskell compiler is a set of pure mathematical transformations. This results in much better high-level optimization facilities. Moreover, pure mathematical computations can be much more easily divided into several threads that may be executed in parallel, which is increasingly important in these days of multi-core CPUs. Finally, pure computations are less error-prone and easier to verify, which adds to Haskell's robustness and to the speed of program development using Haskell.

Haskell purity allows compiler to call only functions whose results are really required to calculate final value of high-level function (i.e., main) - this is called lazy evaluation. It's great thing for pure mathematical computations, but how about I/O actions? Function like:

```
putStrLn "Press any key to begin formatting"
```

can't return any meaningful result value, so how can we ensure that compiler will not omit or reorder its execution? And in general: how we can work with stateful algorithms and side effects in an entirely lazy language? This question has had many different solutions proposed in 18 years of Haskell development (see History of Haskell), though a solution based on monads is now the standard.

2 What is a monad?

What is a monad? It's something from mathematical category theory, which I don't know anymore. In order to understand how monads are used to solve the problem of I/O and side effects, you don't need to know it. It's enough to just know elementary mathematics, like I do.

Let's imagine that we want to implement in Haskell the well-known 'getchar' function. What type should it have? Let's try:

```
getchar :: Char

get2chars = [getchar,getchar]
```

What will we get with 'getchar' having just the 'Char' type? You can see all the possible problems in the definition of 'get2chars':

1. Because the Haskell compiler treats all functions as pure (not having side effects), it can avoid "excessive" calls to 'getchar' and use one returned value twice.
2. Even if it does make two calls, there is no way to determine which call should be performed first. Do you want to return the two chars in the order in which they were read, or in the opposite order? Nothing in the definition of 'get2chars' answers this question.

How can these problems be solved, from the programmer's viewpoint? Let's introduce a fake parameter of 'getchar' to make each call "different" from the compiler's point of view:

```
getchar :: Int -> Char

get2chars = [getchar 1, getchar 2]
```

Right away, this solves the first problem mentioned above - now the compiler will make two calls because it sees them as having different parameters. The whole 'get2chars' function should also have a fake parameter, otherwise we will have the same problem calling it:

```
getchar   :: Int -> Char
get2chars :: Int -> String

get2chars _ = [getchar 1, getchar 2]
```

Now we need to give the compiler some clue to determine which function it should call first. The Haskell language doesn't provide any way to express order of evaluation... except for data dependencies! How about adding an artificial data dependency which prevents evaluation of the second 'getchar' before the first one? In order to achieve this, we will return an additional fake result from 'getchar' that will be used as a parameter for the next 'getchar' call:

```
getchar :: Int -> (Char, Int)

get2chars _ = [a,b]  where (a,i) = getchar 1
                           (b,_) = getchar i
```

So far so good - now we can guarantee that 'a' is read before 'b' because reading 'b' needs the value ('i') that is returned by reading 'a'!

We've added a fake parameter to 'get2chars' but the problem is that the Haskell compiler is too smart! It can believe that the external 'getchar' function is really dependent on its parameter but for 'get2chars' it will see that we're just cheating because we throw it away! Therefore it won't feel obliged to execute the calls in the order we want. How can we fix this? How about passing this fake parameter to the 'getchar' function?! In this case the compiler can't guess that it is really unused.

```
get2chars i0 = [a,b]  where (a,i1) = getchar i0
                            (b,i2) = getchar i1
```

And more - 'get2chars' has all the same purity problems as the 'getchar' function. If you need to call it two times, you need a way to describe the order of these calls. Look at:

```
get4chars = [get2chars 1, get2chars 2]  -- order of 'get2chars' calls isn't defined
```

We already know how to deal with these problems - 'get2chars' should also return some fake value that can be used to order calls:

```
get2chars :: Int -> (String, Int)

get4chars i0 = (a++b)  where (a,i1) = get2chars i0
                             (b,i2) = get2chars i1
```

But what's the fake value 'get2chars' should return? If we use some integer constant, the excessively-smart Haskell compiler will guess that we're cheating again. What about returning the value returned by 'getchar'? See:

```
get2chars :: Int -> (String, Int)
get2chars i0 = ([a,b], i2)  where (a,i1) = getchar i0
                                  (b,i2) = getchar i1
```

Believe it or not, but we've just constructed the whole "monadic" Haskell I/O system.

3 Welcome to the RealWorld, baby

Warning: The following story about IO is incorrect in that it cannot actually explain some important aspects of IO (including interaction and concurrency). However, some people find it useful to begin developing an understanding.

The 'main' Haskell function has the type:

```
main :: RealWorld -> ((), RealWorld)
```

where 'RealWorld' is a fake type used instead of our Int. It's something like the baton passed in a relay race. When 'main' calls some IO function, it passes the "RealWorld" it received as a parameter. All IO functions have similar types involving RealWorld as a parameter and result. To be exact, "IO" is a type synonym defined in the following way:

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

So, 'main' just has type "IO ()", 'getChar' has type "IO Char" and so on. You can think of the type "IO Char" as meaning "take the current RealWorld, do something to it, and return a Char and a (possibly changed) RealWorld". Let's look at 'main' calling 'getChar' two times:

```
getChar :: RealWorld -> (Char, RealWorld)

main :: RealWorld -> ((), RealWorld)
main world0 = let (a, world1) = getChar world0
                  (b, world2) = getChar world1
              in ((), world2)
```

Look at this closely: 'main' passes the "world" it received to the first 'getChar'. This 'getChar' returns some new value of type RealWorld that gets used in the next call. Finally, 'main' returns the "world" it got from the second 'getChar'.

1. Is it possible here to omit any call of 'getChar' if the Char it read is not used? No, because we need to return the "world" that is the result of the second 'getChar' and this in turn requires the "world" returned from the first 'getChar'.
2. Is it possible to reorder the 'getChar' calls? No: the second 'getChar' can't be called before the first one because it uses the "world" returned from the first call.
3. Is it possible to duplicate calls? In Haskell semantics - yes, but real compilers never duplicate work in such simple cases (otherwise, the programs generated will not have any speed guarantees).

As we already said, RealWorld values are used like a baton which gets passed between all routines called by 'main' in strict order. Inside each routine called, RealWorld values are used in the same way. Overall, in order to "compute" the world to be returned from 'main', we should perform each IO procedure that is called from 'main', directly or indirectly. This means that each procedure inserted in the chain will be performed just at the moment (relative to the other IO actions) when we intended it to be called. Let's consider the following program:

```
main = do a <- ask "What is your name?"
          b <- ask "How old are you?"
          return ()

ask s = do putStr s
           readLn
```

Now you have enough knowledge to rewrite it in a low-level way and check that each operation that should be performed will really be performed with the arguments it should have and in the order we expect.

But what about conditional execution? No problem. Let's define the well-known 'when' operation:

```
when :: Bool -> IO () -> IO ()
when condition action world =
    if condition
      then action world
      else ((), world)
```

As you can see, we can easily include or exclude from the execution chain IO procedures (actions) depending on the data values. If 'condition' will be False on the call of 'when', 'action' will never be called because real Haskell compilers, again, never call functions whose results are not required to calculate the final result (*i.e.*, here, the final "world" value of 'main').

Loops and more complex control structures can be implemented in the same way. Try it as an exercise!

Finally, you may want to know how much passing these RealWorld values around the program costs. It's free! These fake values exist solely for the compiler while it analyzes and optimizes the code, but when it gets to assembly code generation, it "suddenly" realize that this type is like "()", so all these parameters and result values can be omitted from the final generated code. Isn't it beautiful?

4 '>>=' and 'do' notation

All beginners (including me) start by thinking that 'do' is some magic statement

that executes IO actions. That's wrong - 'do' is just syntactic sugar that simplifies the writing of procedures that use IO (and also other monads, but that's beyond the scope of this tutorial). 'do' notation eventually gets translated to statements passing "world" values around like we've manually written above and is used to simplify the gluing of several IO actions together. You don't need to use 'do' for just one statement; for instance,

```
  main = do putStr "Hello!"
```

is desugared to:

```
  main = putStr "Hello!"
```

Let's examine how to desugar a 'do' with multiple statements in the following example:

```
main = do putStr "What is your name?"
          putStr "How old are you?"
          putStr "Nice day!"
```

The 'do' statement here just joins several IO actions that should be performed sequentially. It's translated to sequential applications of one of the so-called "binding operators", namely '>>':

```
main = (putStr "What is your name?")
       >> ( (putStr "How old are you?")
            >> (putStr "Nice day!")
          )
```

This binding operator just combines two IO actions, executing them sequentially by passing the "world" between them:

```
(>>) :: IO a -> IO b -> IO b
(action1 >> action2) world0 =
   let (a, world1) = action1 world0
       (b, world2) = action2 world1
   in (b, world2)
```

If defining operators this way looks strange to you, read this definition as follows:

```
action1 >> action2 = action
  where
    action world0 = let (a, world1) = action1 world0
                        (b, world2) = action2 world1
                    in (b, world2)
```

Now you can substitute the definition of '>>' at the places of its usage and check that program constructed by the 'do' desugaring is actually the same as we could write by manually manipulating "world" values.

A more complex example involves the binding of variables using "<-":

```
main = do a <- readLn
          print a
```

This code is desugared into:

```
main = readLn
       >>= (\a -> print a)
```

As you should remember, the '>>' binding operator silently ignores the value of its first action and returns as an overall result the result of its second action only. On the other hand, the '>>=' binding operator (note the extra '=' at the end) allows us to use the result of its first action - it gets passed as an additional parameter to the second one! Look at the definition:

```
(>>=) :: IO a -> (a -> IO b) -> IO b
(action1 >>= action2) world0 =
   let (a, world1) = action1 world0
       (b, world2) = action2 a world1
   in (b, world2)
```

First, what does the type of the second "action" (more precisely, a function which returns an IO action), namely "a -> IO b", mean? By substituting the "IO" definition, we get "a -> RealWorld -> (b, RealWorld)". This means that second action actually has two parameters - the type 'a' actually used inside it, and the value of type RealWorld used for sequencing of IO actions. That's always the case - any IO procedure has one more parameter compared to what you see in its type signature. This parameter is hidden inside the definition of the type alias "IO".

Second, you can use these '>>' and '>>=' operations to simplify your program. For example, in the code above we don't need to introduce the variable, because the result of 'readLn' can be send directly to 'print':

```
main = readLn >>= print
```

And third - as you see, the notation:

```
 do x <- action1
    action2
```

where 'action1' has type "IO a" and 'action2' has type "IO b", translates into:

```
 action1 >>= (\x -> action2)
```

where the second argument of '>>=' has the type "a -> IO b". It's the way the '<-' binding is processed - the name on the left-hand side of '<-' just becomes a parameter of subsequent operations represented as one large IO action. Note also that if 'action1' has type "IO a" then 'x' will just have type "a"; you can think of the effect of '<-' as "unpacking" the IO value of 'action1' into 'x'. Note also that '<-' is

not a true operator; it's pure syntax, just like 'do' itself. Its meaning results only from the way it gets desugared.

Look at the next example:

```
main = do putStr "What is your name?"
          a <- readLn
          putStr "How old are you?"
          b <- readLn
          print (a,b)
```

This code is desugared into:

```
main = putStr "What is your name?"
       >> readLn
       >>= \a -> putStr "How old are you?"
       >> readLn
       >>= \b -> print (a,b)
```

I omitted the parentheses here; both the '>>' and the '>>=' operators are left-associative, but lambda-bindings always stretches as far to the right as possible, which means that the 'a' and 'b' bindings introduced here are valid for all remaining actions. As an exercise, add the parentheses yourself and translate this procedure into the low-level code that explicitly passes "world" values. I think it should be enough to help you finally realize how the 'do' translation and binding operators work.

Oh, no! I forgot the third monadic operator - 'return'. It just combines its two parameters - the value passed and "world":

```
return :: a -> IO a
return a world0  =  (a, world0)
```

How about translating a simple example of 'return' usage? Say,

```
main = do a <- readLn
          return (a*2)
```

Programmers with an imperative language background often think that 'return' in Haskell, as in other languages, immediately returns from the IO procedure. As you can see in its definition (and even just from its type!), such an assumption is totally wrong. The only purpose of using 'return' is to "lift" some value (of type 'a') into the result of a whole action (of type "IO a") and therefore it should generally be used only as the last executed statement of some IO sequence. For example try to translate the following procedure into the corresponding low-level code:

```
main = do a <- readLn
          when (a>=0) $ do
              return ()
          print "a is negative"
```

and you will realize that the 'print' statement is executed even for non-negative values of 'a'. If you need to escape from the middle of an IO procedure, you can use the 'if' statement:

```
main = do a <- readLn
          if (a>=0)
            then return ()
            else print "a is negative"
```

Moreover, Haskell layout rules allow us to use the following layout:

```
main = do a <- readLn
          if (a>=0) then return ()
            else do
          print "a is negative"
          ...
```

that may be useful for escaping from the middle of a longish 'do' statement.

Last exercise: implement a function 'liftM' that lifts operations on plain values to the operations on monadic ones. Its type signature:

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

If that's too hard for you, start with the following high-level definition and rewrite it in low-level fashion:

```
liftM f action = do x <- action
                    return (f x)
```

5 Mutable data (references, arrays, hash tables...)

As you should know, every name in Haskell is bound to one fixed (immutable) value. This greatly simplifies understanding algorithms and code optimization, but it's inappropriate in some cases. As we all know, there are plenty of algorithms that are simpler to implement in terms of updatable variables, arrays and so on. This means that the value associated with a variable, for example, can be different at different execution points, so reading its value can't be considered as a pure function. Imagine, for example, the following code:

```
main = do let a0 = readVariable varA
              _  = writeVariable varA 1
              a1 = readVariable varA
          print (a0, a1)
```

Does this look strange? First, the two calls to 'readVariable' look the same, so the compiler can just reuse the value returned by the first call. Second, the result of the 'writeVariable' call isn't used so the compiler can (and will!) omit this call

completely. To complete the picture, these three calls may be rearranged in any order because they appear to be independent of each other. This is obviously not what was intended. What's the solution? You already know this - use IO actions! Using IO actions guarantees that:

1. the execution order will be retained as written
2. each action will have to be executed
3. the result of the "same" action (such as "readVariable varA") will not be reused

So, the code above really should be written as:

```
import Data.IORef
main = do varA <- newIORef 0  -- Create and initialize a new variable
          a0 <- readIORef varA
          writeIORef varA 1
          a1 <- readIORef varA
          print (a0, a1)
```

Here, 'varA' has the type "IORef Int" which means "a variable (reference) in the IO monad holding a value of type Int". newIORef creates a new variable (reference) and returns it, and then read/write actions use this reference. The value returned by the "readIORef varA" action depends not only on the variable involved but also on the moment this operation is performed so it can return different values on each call.

Arrays, hash tables and any other _mutable_ data structures are defined in the same way - for each of them, there's an operation that creates new "mutable values" and returns a reference to it. Then special read and write operations in the IO monad are used. The following code shows an example using mutable arrays:

```
import Data.Array.IO
main = do arr <- newArray (1,10) 37 :: IO (IOArray Int Int)
          a <- readArray arr 1
          writeArray arr 1 64
          b <- readArray arr 1
          print (a, b)
```

Here, an array of 10 elements with 37 as the initial value at each location is created. After reading the value of the first element (index 1) into 'a' this element's value is changed to 64 and then read again into 'b'. As you can see by executing this code, 'a' will be set to 37 and 'b' to 64.

Other state-dependent operations are also often implemented as IO actions. For example, a random number generator should return a different value on each call. It looks natural to give it a type involving IO:

```
rand :: IO Int
```

Moreover, when you import C routines you should be careful - if this routine is impure, i.e. its result depends on something in the "real world" (file system, memory contents...), internal state and so on, you should give it an IO type. Otherwise, the compiler can "optimize" repetitive calls of this procedure with the same parameters!

For example, we can write a non-IO type for:

```
foreign import ccall
   sin :: Double -> Double
```

because the result of 'sin' depends only on its argument, but

```
foreign import ccall
   tell :: Int -> IO Int
```

If you will declare 'tell' as a pure function (without IO) then you may get the same position on each call!

6 IO actions as values

By this point you should understand why it's impossible to use IO actions inside non-IO (pure) procedures. Such procedures just don't get a "baton"; they don't know any "world" value to pass to an IO action. The RealWorld type is an abstract datatype, so pure functions also can't construct RealWorld values by themselves, and it's a strict type, so 'undefined' also can't be used. So, the prohibition of using IO actions inside pure procedures is just a type system trick (as it usually is in Haskell).

But while pure code can't _execute_ IO actions, it can work with them as with any other functional values - they can be stored in data structures, passed as parameters, returned as results, collected in lists, and partially applied. But an IO action will remain a functional value because we can't apply it to the last argument - of type RealWorld.

In order to _execute_ the IO action we need to apply it to some RealWorld value. That can be done only inside some IO procedure, in its "actions chain". And real execution of this action will take place only when this procedure is called as part of the process of "calculating the final value of world" for 'main'. Look at this example:

```
main world0 = let get2chars = getChar >> getChar
                  ((), world1) = putStr "Press two keys" world0
                  (answer, world2) = get2chars world1
              in ((), world2)
```

Here we first bind a value to 'get2chars' and then write a binding involving

'putStr'. But what's the execution order? It's not defined by the order of the 'let' bindings, it's defined by the order of processing "world" values! You can arbitrarily reorder the binding statements - the execution order will be defined by the data dependency with respect to the "world" values that get passed around. Let's see what this 'main' looks like in the 'do' notation:

```
main = do let get2chars = getChar >> getChar
          putStr "Press two keys"
          get2chars
          return ()
```

As you can see, we've eliminated two of the 'let' bindings and left only the one defining 'get2chars'. The non-'let' statements are executed in the exact order in which they're written, because they pass the "world" value from statement to statement as we described above. Thus, this version of the function is much easier to understand because we don't have to mentally figure out the data dependency of the "world" value.

Moreover, IO actions like 'get2chars' can't be executed directly because they are functions with a RealWorld parameter. To execute them, we need to supply the RealWorld parameter, i.e. insert them in the 'main' chain, placing them in some 'do' sequence executed from 'main' (either directly in the 'main' function, or indirectly in an IO function called from 'main'). Until that's done, they will remain like any function, in partially evaluated form. And we can work with IO actions as with any other functions - bind them to names (as we did above), save them in data structures, pass them as function parameters and return them as results - and they won't be performed until you give them the magic RealWorld parameter!

6.1 Example: a list of IO actions

Let's try defining a list of IO actions:

```
ioActions :: [IO ()]
ioActions = [(print "Hello!"),
             (putStr "just kidding"),
             (getChar >> return ())
            ]
```

I used additional parentheses around each action, although they aren't really required. If you still can't believe that these actions won't be executed immediately, just recall the real type of this list:

```
ioActions :: [RealWorld -> ((), RealWorld)]
```

Well, now we want to execute some of these actions. No problem, just insert them into the 'main' chain:

```
main = do head ioActions
```

```
          ioActions !! 1
          last ioActions
```

Looks strange, right? Really, any IO action that you write in a 'do' statement (or use as a parameter for the '>>'/'>>=' operators) is an expression returning a result of type 'IO a' for some type 'a'. Typically, you use some function that has the type 'x -> y -> ... -> IO a' and provide all the x, y, etc. parameters. But you're not limited to this standard scenario - don't forget that Haskell is a functional language and you're free to compute the functional value required (recall that "IO a" is really a function type) in any possible way. Here we just extracted several functions from the list - no problem. This functional value can also be constructed on-the-fly, as we've done in the previous example - that's also OK. Want to see this functional value passed as a parameter? Just look at the definition of 'when'. Hey, we can buy, sell, and rent these IO actions just like we can with any other functional values! For example, let's define a function that executes all the IO actions in the list:

```
sequence_ :: [IO a] -> IO ()
sequence_ [] = return ()
sequence_ (x:xs) = do x
                      sequence_ xs
```

No black magic - we just extract IO actions from the list and insert them into a chain of IO operations that should be performed one after another (in the same order that they occurred in the list) to "compute the final world value" of the entire 'sequence_' call.

With the help of 'sequence_', we can rewrite our last 'main' function as:

```
main = sequence_ ioActions
```

Haskell's ability to work with IO actions as with any other (functional and non-functional) values allows us to define control structures of arbitrary complexity. Try, for example, to define a control structure that repeats an action until it returns the 'False' result:

```
while :: IO Bool -> IO ()
while action = ???
```

Most programming languages don't allow you to define control structures at all, and those that do often require you to use a macro-expansion system. In Haskell, control structures are just trivial functions anyone can write.

6.2 Example: returning an IO action as a result

How about returning an IO action as the result of a function? Well, we've done

this each time we've defined an IO procedure - they all return IO actions that need a RealWorld value to be performed. While we usually just execute them as part of a higher-level IO procedure, it's also possible to just collect them without actual execution:

```
main = do let a = sequence ioActions
              b = when True getChar
              c = getChar >> getChar
          putStr "These 'let' statements are not executed!"
```

These assigned IO procedures can be used as parameters to other procedures, or written to global variables, or processed in some other way, or just executed later, as we did in the example with 'get2chars'.

But how about returning a parameterized IO action from an IO procedure? Let's define a procedure that returns the i'th byte from a file represented as a Handle:

```
readi h i = do hSeek h AbsoluteSeek i
               hGetChar h
```

So far so good. But how about a procedure that returns the i'th byte of a file with a given name without reopening it each time?

```
readfilei :: String -> IO (Integer -> IO Char)
readfilei name = do h <- openFile name ReadMode
                    return (readi h)
```

As you can see, it's an IO procedure that opens a file and returns... another IO procedure that will read the specified byte. But we can go further and include the 'readi' body in 'readfilei':

```
readfilei name = do h <- openFile name ReadMode
                    let readi h i = do hSeek h AbsoluteSeek i
                                       hGetChar h
                    return (readi h)
```

That's a little better. But why do we add 'h' as a parameter to 'readi' if it can be obtained from the environment where 'readi' is now defined? An even shorter version is this:

```
readfilei name = do h <- openFile name ReadMode
                    let readi i = do hSeek h AbsoluteSeek i
                                     hGetChar h
                    return readi
```

What have we done here? We've build a parameterized IO action involving local names inside 'readfilei' and returned it as the result. Now it can be used in the following way:

```
main = do myfile <- readfilei "test"
          a <- myfile 0
          b <- myfile 1
          print (a,b)
```

This way of using IO actions is very typical for Haskell programs - you just construct one or more IO actions that you need, with or without parameters, possibly involving the parameters that your "constructor" received, and return them to the caller. Then these IO actions can be used in the rest of the program without any knowledge about your internal implementation strategy. One thing this can be used for is to partially emulate the OOP (or more precisely, the ADT) programming paradigm.

6.3 Example: a memory allocator generator

As an example, one of my programs has a module which is a memory suballocator. It receives the address and size of a large memory block and returns two procedures - one to allocate a subblock of a given size and the other to free the allocated subblock:

```
memoryAllocator :: Ptr a -> Int -> IO (Int -> IO (Ptr b),
                                       Ptr c -> IO ())

memoryAllocator buf size = do ......
                              let alloc size = do ...
                                                  ...
                                  free ptr = do ...
                                                ...
                              return (alloc, free)
```

How this is implemented? 'alloc' and 'free' work with references created inside the memoryAllocator procedure. Because the creation of these references is a part of the memoryAllocator IO actions chain, a new independent set of references will be created for each memory block for which memoryAllocator is called:

```
memoryAllocator buf size = do start <- newIORef buf
                              end <- newIORef (buf `plusPtr` size)
                              ...
```

These two references are read and written in the 'alloc' and 'free' definitions (we'll implement a very simple memory allocator for this example):

```
    ...
    let alloc size = do addr <- readIORef start
                        writeIORef start (addr `plusPtr` size)
                        return addr

    let free ptr = do writeIORef start ptr
```

What we've defined here is just a pair of closures that use state available at the moment of their definition. As you can see, it's as easy as in any other functional language, despite Haskell's lack of direct support for impure functions.

The following example uses procedures, returned by memoryAllocator, to

simultaneously allocate/free blocks in two independent memory buffers:

```
main = do buf1 <- mallocBytes (2^16)
          buf2 <- mallocBytes (2^20)
          (alloc1, free1) <- memoryAllocator buf1 (2^16)
          (alloc2, free2) <- memoryAllocator buf2 (2^20)
          ptr11 <- alloc1 100
          ptr21 <- alloc2 1000
          free1 ptr11
          free2 ptr21
          ptr12 <- alloc1 100
          ptr22 <- alloc2 1000
```

6.4 Example: emulating OOP with record types

Let's implement the classical OOP example: drawing figures. There are figures of different types: circles, rectangles and so on. The task is to create a heterogeneous list of figures. All figures in this list should support the same set of operations: draw, move and so on. We will represent these operations as IO procedures. Instead of a "class" let's define a structure containing implementations of all the procedures required:

```
data Figure = Figure { draw :: IO (),
                       move :: Displacement -> IO ()
                     }

type Displacement = (Int, Int)  -- horizontal and vertical displacement in points
```

The constructor of each figure's type should just return a Figure record:

```
circle    :: Point -> Radius -> IO Figure
rectangle :: Point -> Point -> IO Figure

type Point = (Int, Int)  -- point coordinates
type Radius = Int        -- circle radius in points
```

We will "draw" figures by just printing their current parameters. Let's start with a simplified implementation of the 'circle' and 'rectangle' constructors, without actual 'move' support:

```
circle center radius = do
    let description = "  Circle at "++show center++" with radius "++show radius
    return $ Figure { draw = putStrLn description }

rectangle from to = do
    let description = "  Rectangle "++show from++"-"++show to)
    return $ Figure { draw = putStrLn description }
```

As you see, each constructor just returns a fixed 'draw' procedure that prints parameters with which the concrete figure was created. Let's test it:

```
drawAll :: [Figure] -> IO ()
drawAll figures = do putStrLn "Drawing figures:"
                     mapM_ draw figures

main = do figures <- sequence [circle (10,10) 5,
                               circle (20,20) 3,
                               rectangle (10,10) (20,20),
                               rectangle (15,15) (40,40)]
          drawAll figures
```

Now let's define "full-featured" figures that can actually be moved around. In order to achieve this, we should provide each figure with a mutable variable that holds each figure's current screen location. The type of this variable will be "IORef Point". This variable should be created in the figure constructor and manipulated in IO procedures (closures) enclosed in the Figure record:

```
circle center radius = do
    centerVar <- newIORef center

    let drawF = do center <- readIORef centerVar
                   putStrLn ("  Circle at "++show center
                             ++" with radius "++show radius)

    let moveF (addX,addY) = do (x,y) <- readIORef centerVar
                               writeIORef centerVar (x+addX, y+addY)

    return $ Figure { draw=drawF, move=moveF }


rectangle from to = do
    fromVar <- newIORef from
    toVar   <- newIORef to

    let drawF = do from <- readIORef fromVar
                   to   <- readIORef toVar
                   putStrLn ("  Rectangle "++show from++"-"++show to)

    let moveF (addX,addY) = do (fromX,fromY) <- readIORef fromVar
                               (toX,toY)     <- readIORef toVar
                               writeIORef fromVar (fromX+addX, fromY+addY)
                               writeIORef toVar   (toX+addX, toY+addY)

    return $ Figure { draw=drawF, move=moveF }
```

Now we can test the code which moves figures around:

```
main = do figures <- sequence [circle (10,10) 5,
                               rectangle (10,10) (20,20)]
          drawAll figures
          mapM_ (\fig -> move fig (10,10)) figures
          drawAll figures
```

It's important to realize that we are not limited to including only IO actions in a record that's intended to simulate a C++/Java-style interface. The record can also include values, IORefs, pure functions - in short, any type of data. For example, we can easily add to the Figure interface fields for area and origin:

```
data Figure = Figure { draw :: IO (),
                       move :: Displacement -> IO (),
                       area :: Double,
                       origin :: IORef Point
                     }
```

7 Exception handling (under development)

Although Haskell provides a set of exception raising/handling features comparable to those in popular OOP languages (C++, Java, C#), this part of the language receives much less attention. This is for two reasons. First, you just don't need to worry as much about them - most of the time it just works "behind the scenes". The second reason is that Haskell, lacking OOP inheritance, doesn't allow the programmer to easily subclass exception types, therefore limiting flexibility of exception handling.

The Haskell RTS raises more exceptions than traditional languages - pattern match failures, calls with invalid arguments (such as **head []**) and computations whose results depend on special values **undefined** and **error "...."** all raise their own exceptions:

example 1:

```
main = print (f 2)

f 0 = "zero"
f 1 = "one"
```

example 2:

```
main = print (head [])
```

example 3:

```
main = print (1 + (error "Value that wasn't initialized or cannot be computed"))
```

This allows to write programs in much more error-prone way.

8 Interfacing with C/C++ and foreign libraries (under development)

While Haskell is great at algorithm development, speed isn't its best side. We can combine the best of both worlds, though, by writing speed-critical parts of program in C and rest in Haskell. We just need a way to call C functions from Haskell and vice versa, and to marshal data between two worlds.

We also need to interact with C world for using Windows/Linux APIs, linking to various libraries and DLLs. Even interfacing with other languages requires to go through C world as "common denominator". Chapter 8 of the Haskell 2010 report (https://www.haskell.org/onlinereport/haskell2010/haskellch8.html) provides a complete description of interfacing with C.

We will learn FFI via a series of examples. These examples include C/C++ code, so they need C/C++ compilers to be installed, the same will be true if you need to include code written in C/C++ in your program (C/C++ compilers are not required when you just need to link with existing libraries providing APIs with C calling convention). On Unix (and Mac OS?) systems, the system-wide default C/C++ compiler is typically used by GHC installation. On Windows, no default compilers exist, so GHC is typically shipped with a C compiler, and you may find on the download page a GHC distribution bundled with C and C++ compilers. Alternatively, you may find and install a GCC/MinGW version compatible with your GHC installation.

If you need to make your C/C++ code as fast as possible, you may compile your code by Intel compilers instead of GCC. However, these compilers are not free, moreover on Windows, code compiled by Intel compilers may not interact correctly with GHC-compiled code, unless one of them is put into DLLs (due to object file incompatibility).

More links (http://www.haskell.org/haskellwiki/Applications_and_libraries/Interfacing_other_languages) :

C->Haskell (http://www.cse.unsw.edu.au/~chak/haskell/c2hs/)
: A lightweight tool for implementing access to C libraries from Haskell.

HSFFIG
: Haskell FFI Binding Modules Generator (HSFFIG) is a tool that takes a C library include file (.h) and generates Haskell Foreign Functions Interface import declarations for items (functions, structures, etc.) the header defines.

MissingPy (http://quux.org/devel/missingpy)
: MissingPy is really two libraries in one. At its lowest level, MissingPy is a library designed to make it easy to call into Python from Haskell. It provides full support for interpreting arbitrary Python code, interfacing with a good part of the Python/C API, and handling Python objects. It also provides tools for converting between Python objects and their Haskell equivalents. Memory management is handled for you, and Python exceptions get mapped to Haskell Dynamic exceptions. At a higher level, MissingPy contains Haskell

interfaces to some Python modules.

HsLua
: A Haskell interface to the Lua scripting language

8.1 Calling functions

First, we will learn how to call C functions from Haskell and Haskell functions from C. The first example consists of three files:

main.hs:

```
{-# LANGUAGE ForeignFunctionInterface #-}

main = do print "Hello from main"
          c_function

haskell_function = print "Hello from haskell_function"

foreign import ccall safe "prototypes.h"
    c_function :: IO ()

foreign export ccall
    haskell_function :: IO ()
```

evil.c:

```
#include <stdio.h>
#include "prototypes.h"

void c_function (void)
{
  printf("Hello from c_function\n");
  haskell_function();
}
```

prototypes.h:

```
extern void c_function (void);
extern void haskell_function (void);
```

It may be compiled and linked in one step by ghc:

```
ghc --make main.hs evil.c
```

Or, you may compile C module(s) separately and link in .o files (this may be preferable if you use `make` and don't want to recompile unchanged sources; ghc's --make option provides smart recompilation only for .hs files):

```
ghc -c evil.c
```

```
ghc --make main.hs evil.o
```

You may use gcc/g++ directly to compile your C/C++ files but I recommend to do linking via ghc because it adds a lot of libraries required for execution of Haskell code. For the same reason, even if your main routine is written in C/C++, I recommend calling it from the Haskell function
`main`
- otherwise you'll have to explicitly init/shutdown the GHC RTS (run-time system).

We use the "foreign import" specification to import foreign routines into our Haskell world, and "foreign export" to export Haskell routines into the external world. Note that the import statement creates a new Haskell symbol (from the external one), while the export statement uses a Haskell symbol previously defined. Technically speaking, both types of statements create a wrapper that converts the names and calling conventions from C to Haskell or vice versa.

8.2 All about the "foreign" statement

The "ccall" specifier in foreign statements means the use of C (not C++ !) calling convention. This means that if you want to write the external function in C++ (instead of C) you should add **export "C"** specification to its declaration - otherwise you'll get linking errors. Let's rewrite our first example to use C++ instead of C:

prototypes.h:

```
#ifdef __cplusplus
extern "C" {
#endif

extern void c_function (void);
extern void haskell_function (void);

#ifdef __cplusplus
}
#endif
```

Compile it via:

```
ghc --make main.hs evil.cpp
```

where evil.cpp is just a renamed copy of evil.c from the first example. Note that the new prototypes.h is written to allow compiling it both as C and C++ code. When it's included from evil.cpp, it's compiled as C++ code. When GHC compiles main.hs via the C compiler (enabled by -fvia-C option), it also includes prototypes.h but compiles it in C mode. It's why you need to specify .h files in "foreign" declarations - depending on which Haskell compiler you use, these files may be included to check consistency of C and Haskell declarations.

The quoted part of the foreign statement may also be used to import or export a function under another name--for example,

```
foreign import ccall safe "prototypes.h CFunction"
    c_function :: IO ()

foreign export ccall "HaskellFunction"
    haskell_function :: IO ()
```

specifies that the C function called CFunction will become known as the Haskell function c_function, while the Haskell function haskell_function will be known in the C world as HaskellFunction. It's required when the C name doesn't conform to Haskell naming requirements.

Although the Haskell FFI standard tells about many other calling conventions in addition to ccall (e.g. cplusplus, jvm, net) current Haskell implementations support only ccall and stdcall. The latter, also called the "Pascal" calling convention, is used to interface with WinAPI:

```
foreign import stdcall unsafe "windows.h SetFileApisToOEM"
  setFileApisToOEM :: IO ()
```

And finally, about the safe/unsafe specifier: a C function imported with the "unsafe" keyword is called directly and the Haskell runtime is stopped while the C function is executed (when there are several OS threads executing the Haskell program, only the current OS thread is delayed). This call doesn't allow recursively entering into the Haskell world by calling any Haskell function - the Haskell RTS is just not prepared for such an event. However, unsafe calls are as quick as calls in C world. It's ideal for "momentary" calls that quickly return back to the caller.

When "safe" is specified, the C function is called in safe environment - the Haskell execution context is saved, so it's possible to call back to Haskell and, if the C call takes a long time, another OS thread may be started to execute Haskell code (of course, in threads other than the one that called the C code). This has its own price, though - around 1000 CPU ticks per call.

You can read more about interaction between FFI calls and Haskell concurrency in [7].

8.3 Marshalling simple types

Calling by itself is relatively easy; the real problem of interfacing languages with different data models is passing data between them. In this case, there is no guarantee that Haskell's Int is represented in memory the same way as C's int, nor Haskell's Double the same as C's double and so on. While on *some* platforms they are the same and you can write throw-away programs relying on these, the goal of portability requires you to declare imported and exported

functions using special types described in the FFI standard, which are guaranteed to correspond to C types. These are:

```
import Foreign.C.Types (               -- equivalent to the following C type:
         CChar, CUChar,                --  char/unsigned char
         CShort, CUShort,              --  short/unsigned short
         CInt, CUInt, CLong, CULong,   --  int/unsigned/long/unsigned long
         CFloat, CDouble...)           --  float/double
```

Now we can import and export typeful C/Haskell functions:

```
foreign import ccall unsafe "math.h"
    c_sin :: CDouble -> CDouble
```

Note that pure C functions (those whose results depend only on their arguments) are imported without IO in their return type. The "const" specifier in C is not reflected in Haskell types, so appropriate compiler checks are not performed.

All these numeric types are instances of the same classes as their Haskell cousins (Ord, Num, Show and so on), so you may perform calculations on these data directly. Alternatively, you may convert them to native Haskell types. It's very typical to write simple wrappers around imported and exported functions just to provide interfaces having native Haskell types:

```
-- |Type-conversion wrapper around c_sin
sin :: Double -> Double
sin = fromRational . c_sin . toRational
```

8.4 Memory management

8.5 Marshalling strings

```
import Foreign.C.String (    -- representation of strings in C
         CString,            -- = Ptr CChar
         CStringLen)         -- = (Ptr CChar, Int)
foreign import ccall unsafe "string.h"
    c_strlen :: CString -> IO CSize     -- CSize defined in Foreign.C.Types and is equal to s
-- |Type-conversion wrapper around c_strlen
strlen :: String -> Int
strlen = ....
```

8.6 Marshalling composite types

A C array may be manipulated in Haskell as StorableArray (http://haskell.org/haskellwiki/Arrays#StorableArray_.28module_Data.Array.Storable.29) .

There is no built-in support for marshalling C structures and using C constants in Haskell. These are implemented in c2hs preprocessor, though.

Binary marshalling (serializing) of data structures of any complexity is implemented in library Binary.

8.7 Dynamic calls

8.8 DLLs

because i don't have experience of using DLLs, can someone write into this section? ultimately, we need to consider the following tasks:

- using DLLs of 3rd-party libraries (such as ziplib)
- putting your own C code into a DLL to use in Haskell
- putting Haskell code into a DLL which may be called from C code

9 Dark side of IO monad

9.1 unsafePerformIO

Programmers coming from an imperative language background often look for a way to execute IO actions inside a pure procedure. But what does this mean? Imagine that you're trying to write a procedure that reads the contents of a file with a given name, and you try to write it as a pure (non-IO) function:

```
readContents :: Filename -> String
```

Defining readContents as a pure function will certainly simplify the code that uses it. But it will also create problems for the compiler:

1. This call is not inserted in a sequence of "world transformations", so the compiler doesn't know at what exact moment you want to execute this action. For example, if the file has one kind of contents at the beginning of the program and another at the end - which contents do you want to see? You have no idea when (or even if) this function is going to get invoked, because Haskell sees this function as pure and feels free to reorder the execution of any or all pure functions as needed.
2. Attempts to read the contents of files with the same name can be factored (*i.e.* reduced to a single call) despite the fact that the file (or the current directory) can be changed between calls. Again, Haskell considers all non-IO functions to be pure and feels free to omit multiple calls with the same parameters.

So, implementing pure functions that interact with the Real World is considered to be Bad Behavior. Good boys and girls never do it ;)

Nevertheless, there are (semi-official) ways to use IO actions inside of pure functions. As you should remember this is prohibited by requiring the RealWorld "baton" in order to call an IO action. Pure functions don't have the baton, but there is a special "magic" procedure that produces this baton from nowhere, uses

it to call an IO action and then throws the resulting "world" away! It's a little low-level magic. This very special (and dangerous) procedure is:

```
unsafePerformIO :: IO a -> a
```

Let's look at its (possible) definition:

```
unsafePerformIO :: (RealWorld -> (a, RealWorld)) -> a
unsafePerformIO action = let (a, world1) = action createNewWorld
                         in a
```

where 'createNewWorld' is an internal function producing a new value of the RealWorld type.

Using unsafePerformIO, you can easily write pure functions that do I/O inside. But don't do this without a real need, and remember to follow this rule: the compiler doesn't know that you are cheating; it still considers each non-IO function to be a pure one. Therefore, all the usual optimization rules can (and will!) be applied to its execution. So you must ensure that:

1. The result of each call depends only on its arguments.
2. You don't rely on side-effects of this function, which may be not executed if its results are not needed.

Let's investigate this problem more deeply. Function evaluation in Haskell is determined by a value's necessity - the language computes only the values that are really required to calculate the final result. But what does this mean with respect to the 'main' function? To "calculate the final world's" value, you need to perform all the intermediate IO actions that are included in the 'main' chain. By using 'unsafePerformIO' we call IO actions outside of this chain. What guarantee do we have that they will be run at all? None. The only time they will be run is if running them is required to compute the overall function result (which in turn should be required to perform some action in the 'main' chain). This is an example of Haskell's evaluation-by-need strategy. Now you should clearly see the difference:

- An IO action inside an IO procedure is guaranteed to execute as long as it is (directly or indirectly) inside the 'main' chain - even when its result isn't used (because the implicit "world" value it returns *will* be used). You directly specify the order of the action's execution inside the IO procedure. Data dependencies are simulated via the implicit "world" values that are passed from each IO action to the next.

- An IO action inside 'unsafePerformIO' will be performed only if result of this operation is really used. The evaluation order is not guaranteed and you should not rely on it (except when you're sure about whatever data dependencies may exist).

I should also say that inside 'unsafePerformIO' call you can organize a small internal chain of IO actions with the help of the same binding operators and/or 'do' syntactic sugar we've seen above. For example, here's a particularly convoluted way to compute the integer that comes after zero:

```
one :: Int
one = unsafePerformIO $ do var <- newIORef 0
                           modifyIORef var (+1)
                           readIORef var
```

and in this case ALL the operations in this chain will be performed as long as the result of the 'unsafePerformIO' call is needed. To ensure this, the actual 'unsafePerformIO' implementation evaluates the "world" returned by the 'action':

```
unsafePerformIO action = let (a,world1) = action createNewWorld
                         in (world1 `seq` a)
```

(The 'seq' operation strictly evaluates its first argument before returning the value of the second one).

9.2 inlinePerformIO

inlinePerformIO has the same definition as unsafePerformIO but with addition of INLINE pragma:

```
-- | Just like unsafePerformIO, but we inline it. Big performance gains as
-- it exposes lots of things to further inlining
{-# INLINE inlinePerformIO #-}
inlinePerformIO action = let (a, world1) = action createNewWorld
                         in (world1 `seq` a)
#endif
```

Semantically inlinePerformIO = unsafePerformIO in as much as either of those have any semantics at all.

The difference of course is that inlinePerformIO is even less safe than unsafePerformIO. While ghc will try not to duplicate or common up different uses of unsafePerformIO, we aggressively inline inlinePerformIO. So you can really only use it where the IO content is really properly pure, like reading from an immutable memory buffer (as in the case of ByteStrings). However things like allocating new buffers should not be done inside inlinePerformIO since that can easily be floated out and performed just once for the whole program, so you end up with many things sharing the same buffer, which would be bad.

So the rule of thumb is that IO things wrapped in unsafePerformIO have to be externally pure while with inlinePerformIO it has to be really really pure or it'll all go horribly wrong.

That said, here's some really hairy code. This should frighten any pure functional programmer...

```
write :: Int -> (Ptr Word8 -> IO ()) -> Put ()
write !n body = Put $ \c buf@(Buffer fp o u l) ->
  if n <= l
    then write' c fp o u l
    else write' (flushOld c n fp o u) (newBuffer c n) 0 0 0

  where {-# NOINLINE write' #-}
        write' c !fp !o !u !l =
          -- warning: this is a tad hardcore
          inlinePerformIO
            (withForeignPtr fp
              (\p -> body $! (p `plusPtr` (o+u))))
          `seq` c () (Buffer fp o (u+n) (l-n))
```

it's used like:

```
word8 w = write 1 (\p -> poke p w)
```

This does not adhere to my rule of thumb above. Don't ask exactly why we claim it's safe :-) (and if anyone really wants to know, ask Ross Paterson who did it first in the Builder monoid)

9.3 unsafeInterleaveIO

But there is an even stranger operation called 'unsafeInterleaveIO' that gets the "official baton", makes its own pirate copy, and then runs an "illegal" relay-race in parallel with the main one! I can't talk further about its behavior without causing grief and indignation, so it's no surprise that this operation is widely used in countries that are hotbeds of software piracy such as Russia and China! ;) Don't even ask me - I won't say anything more about this dirty trick I use all the time ;)

One can use unsafePerformIO (not unsafeInterleaveIO) to perform I/O operations not in predefined order but by demand. For example, the following code:

```
do let c = unsafePerformIO getChar
   do_proc c
```

will perform getChar I/O call only when value of c is really required by code, i.e. it this call will be performed lazily as any usual Haskell computation.

Now imagine the following code:

```
do let s = [unsafePerformIO getChar, unsafePerformIO getChar, unsafePerformIO getChar]
   do_proc s
```

Three chars inside this list will be computed on demand too, and this means that their values will depend on the order they are consumed. It is not that we usually need.

unsafeInterleaveIO solves this problem - it performs I/O only on demand but allows to define exact *internal* execution order for parts of your datastructure. It is why I wrote that unsafeInterleaveIO makes illegal copy of baton.

First, unsafeInterleaveIO has (IO a) action as a parameter and returns value of type 'a':

```
do str <- unsafeInterleaveIO myGetContents
```

Second, unsafeInterleaveIO don't perform any action immediately, it only creates a box of type 'a' which on requesting this value will perform action specified as a parameter.

Third, this action by itself may compute the whole value immediately or... use unsafeInterleaveIO again to defer calculation of some sub-components:

```
myGetContents = do
   c <- getChar
   s <- unsafeInterleaveIO myGetContents
   return (c:s)
```

This code will be executed only at the moment when value of str is really demanded. In this moment, getChar will be performed (with result assigned to c) and one more lazy IO box will be created - for s. This box again contains link to the myGetContents call

Then, list cell returned that contains one char read and link to myGetContents call as a way to compute rest of the list. Only at the moment when next value in list required, this operation will be performed again

As a final result, we get inability to read second char in list before first one, but lazy character of reading in whole. bingo!

PS: of course, actual code should include EOF checking. also note that you can read many chars/records at each call:

```
myGetContents = do
   c <- replicateM 512 getChar
   s <- unsafeInterleaveIO myGetContents
   return (c++s)
```

10 A safer approach: the ST monad

We said earlier that we can use unsafePerformIO to perform computations that are totally pure but nevertheless interact with the Real World in some way. There is, however, a better way! One that remains totally pure and yet allows the use of references, arrays, and so on -- and it's done using, you guessed it, type magic.

This is the ST monad.

The ST monad's version of unsafePerformIO is called runST, and it has a very unusual type.

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

The s variable in the ST monad is the state type. Moreover, all the fun mutable stuff available in the ST monad is quantified over s:

```
newSTRef :: a -> ST s (STRef s a)
newArray_ :: Ix i => (i, i) -> ST s (STArray s i e)
```

So why does runST have such a funky type? Let's see what would happen if we wrote

```
makeSTRef :: a -> STRef s a
makeSTRef a = runST (newSTRef a)
```

This fails, because newSTRef a doesn't work for all state types s -- it only works for the s from the return type STRef s a.

This is all sort of wacky, but the result is that you can only run an ST computation where the output type is functionally pure, and makes no references to the internal mutable state of the computation. The ST monad doesn't have access to I/O operations like writing to the console, either -- only references, arrays, and suchlike that come in handy for pure computations.

Important note -- the state type doesn't actually mean anything. We never have a value of type s, for instance. It's just a way of getting the type system to do the work of ensuring purity for us, with smoke and mirrors.

It's really just type system magic: secretly, on the inside, runST runs a computation with the real world baton just like unsafePerformIO. Their internal implementations are almost identical: in fact, there's a function

```
stToIO :: ST RealWorld a -> IO a
```

The difference is that ST uses type system magic to forbid unsafe behavior like extracting mutable objects from their safe ST wrapping, but allowing purely functional outputs to be performed with all the handy access to mutable references and arrays.

So here's how we'd rewrite our function using unsafePerformIO from above:

```
oneST :: ST s Int -- note that this works correctly for any s
oneST = do var <- newSTRef 0
           modifySTRef var (+1)
           readSTRef var

one :: Int
```

```
one = runST oneST
```

11 Welcome to the machine: the actual GHC implementation

A little disclaimer: I should say that I'm not describing here exactly what a monad is (I don't even completely understand it myself) and my explanation shows only one _possible_ way to implement the IO monad in Haskell. For example, the hbc Haskell compiler and the Hugs interpreter implements the IO monad via continuations. I also haven't said anything about exception handling, which is a natural part of the "monad" concept. You can read the "All About Monads" guide to learn more about these topics.

But there is some good news: first, the IO monad understanding you've just acquired will work with any implementation and with many other monads. You just can't work with RealWorld values directly.

Second, the IO monad implementation described here is really used in the GHC, yhc/nhc (jhc, too?) compilers. Here is the actual IO definition from the GHC sources:

```
newtype IO a = IO (State# RealWorld -> (# State# RealWorld, a #))
```

It uses the "State# RealWorld" type instead of our RealWorld, it uses the "(# #)" strict tuple for optimization, and it adds an IO data constructor around the type. Nevertheless, there are no significant changes from the standpoint of our explanation. Knowing the principle of "chaining" IO actions via fake "state of the world" values, you can now easily understand and write low-level implementations of GHC I/O operations.

11.1 The Yhc/nhc98 implementation

```
data World = World
newtype IO a = IO (World -> Either IOError a)
```

This implementation makes the "World" disappear somewhat, and returns Either a result of type "a", or if an error occurs then "IOError". The lack of the World on the right-hand side of the function can only be done because the compiler knows special things about the IO type, and won't overoptimise it.

12 Further reading

[1] This tutorial is largely based on the Simon Peyton Jones' paper Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell (http://research.microsoft.com/%7Esimonpj/Papers/marktoberdorf) . I hope that my tutorial improves his original explanation of the Haskell I/O system and brings it closer to the point of view of beginning Haskell programmers. But if you need to learn about concurrency, exceptions and FFI in Haskell/GHC, the original paper is the best source of information.

[2] You can find more information about concurrency, FFI and STM at the GHC/Concurrency#Starting points page.

[3] The Arrays page contains exhaustive explanations about using mutable arrays.

[4] Look also at the Using monads page, which contains tutorials and papers really describing these mysterious monads.

[5] An explanation of the basic monad functions, with examples, can be found in the reference guide A tour of the Haskell Monad functions (http://members.chello.nl/hjgtuyl/tourdemonad.html) , by Henk-Jan van Tuyl.

[6] Official FFI specifications can be found on the page The Haskell 98 Foreign Function Interface 1.0: An Addendum to the Haskell 98 Report (http://www.cse.unsw.edu.au/~chak/haskell/ffi/)

[7] Using FFI in multithreaded programs described in paper Extending the Haskell Foreign Function Interface with Concurrency (http://www.haskell.org/~simonmar/bib/concffi04_abstract.html)

Do you have more questions? Ask in the haskell-cafe mailing list (http://www.haskell.org/mailman/listinfo/haskell-cafe) .

13 To-do list

If you are interested in adding more information to this manual, please add your questions/topics here.

Topics:

- fixIO and 'mdo'
- Q monad

Questions:

- split '>>='/'>>'/return section and 'do' section, more examples of using binding operators
- IORef detailed explanation (==const*), usage examples, syntax sugar, unboxed refs

- explanation of how the actual data "in" mutable references are inside 'RealWorld', rather than inside the references themselves ('IORef','IOArray',&c.)
- control structures developing - much more examples
- unsafePerformIO usage examples: global variable, ByteString, other examples
- how 'unsafeInterLeaveIO' can be seen as a kind of concurrency, and therefore isn't so unsafe (unlike 'unsafeInterleaveST' which really is unsafe)
- discussion about different senses of "safe"/"unsafe" (like breaking equational reasoning vs. invoking undefined behaviour (so can corrupt the run-time system))
- actual GHC implementation - how to write low-level routines on example of newIORef implementation

This manual is collective work, so feel free to add more information to it yourself. The final goal is to collectively develop a comprehensive manual for using the IO monad.

Category:

- Tutorials
