Inside 736-131

Existential Pontification and Generalized Abstract Digressions

Unraveling the mystery of the IO monad

When we teach beginners about Haskell, one of the things we handwave away is how the IO monad works. Yes, it's a monad, and yes, it does IO, but it's not something you can implement in Haskell itself, giving it a somewhat magical quality. In today's post, I'd like to unravel the mystery of the IO monad by describing how GHC implements the IO monad internally in terms of primitive operations and the real world token. After reading this post, you should be able to understand the resolution of this ticket as well as the Core output of this Hello World! program:

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

Nota bene: **This is not a monad tutorial**. This post assumes the reader knows what monads are! However, the first section reviews a critical concept of strictness as applied to monads, because it is critical to the correct functioning of the IO monad.

The lazy and strict State monad

As a prelude to the IO monad, we will briefly review the State monad, which forms the operational basis for the IO monad (the IO monad is implemented as if it were a strict State monad with a *special* form of state, though there are some important differences—that's the magic of it.) If you feel comfortable with the difference between the lazy and strict state monad, you can skip this section. Otherwise, read on. The data type constructor of the State monad is as follows:

```
newtype State s a = State { runState :: s -> (a, s) }
```

A running a computation in the state monad involves giving it some incoming state, and retrieving from it the resulting state and the actual value of the computation. The monadic structure involves *threading* the state through the various computations. For example, this snippet of code in the state monad:

```
do x <- doSomething
   y <- doSomethingElse
   return (x + y)
```

could be rewritten (with the newtype constructor removed) as:

```
\s ->
let (x, s')  = doSomething s
    (y, s'') = doSomethingElse s' in
(x + y, s'')
```

Now, a rather interesting experiment I would like to pose for the reader is this: suppose that `doSomething` and `doSomethingElse` were traced: that is, when evaluated, they outputted a trace message. That is:

```
doSomething s = trace "doSomething" $ ...
doSomethingElse s = trace "doSomethingElse" $ ...
```

Is there ever a situation in which the trace for `doSomethingElse` would fire before `doSomething`, in the case that we forced the result of the elements of this do block? In a strict language, the answer would obviously be no; you have to do each step of the stateful computation in order. But Haskell is lazy, and in another situation it's conceivable that the result of `doSomethingElse` might be requested before `doSomething` is. Indeed, here is such an example of some code:

```
import Debug.Trace

f = \s ->
        let (x, s')  = doSomething s
            (y, s'') = doSomethingElse s'
        in (3, s'')

doSomething s = trace "doSomething" $ (0, s)
doSomethingElse s = trace "doSomethingElse" $ (3, s)

main = print (f 2)
```

What has happened is that we are lazy in the state value, so when we demanded the value of `s''`, we forced `doSomethingElse` and were presented with an indirection to `s'`, which then caused us to force `doSomething`.

Suppose we actually did want `doSomething` to always execute before `doSomethingElse`. In this case, we can fix things up by making our state strict:

```
f = \s ->
        case doSomething s of
            (x, s') -> case doSomethingElse s' of
                          (y, s'') -> (3, s'')
```

This subtle transformation from let (which is lazy) to case (which is strict) lets us now preserve ordering. In fact, it will turn out, we won't be given a choice in the matter: due to how primitives work out we have to do things this way. Keep your eye on the case: it will show up again when we start looking at Core.

Bonus. Interestingly enough, if you use irrefutable patterns, the case-code is equivalent to the original let-code:

```
f = \s ->
        case doSomething s of
            ~(x, s') -> case doSomethingElse s' of
                            ~(y, s'') -> (3, s'')
```

Primitives

The next part of our story are the primitive types and functions provided by GHC. These are the mechanism by which GHC exports types and functionality that would not be normally implementable in Haskell: for example, unboxed types, adding together two 32-bit integers, or doing an IO action (mostly, writing bits to memory locations). They're very GHC specific, and normal Haskell users never see them. In fact, they're so special you need to enable a language extension to use them (the `MagicHash`)! The IO type is constructed with these primitives in `GHC.Types`:

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

In order to understand the `IO` type, we will need to learn about a few of these primitives. But it should be very clear that this looks a lot like the state monad...

The first primitive is the *unboxed tuple*, seen in code as `(# x, y #)`. Unboxed tuples are syntax for a "multiple return" calling convention; they're not actually real tuples and can't be put in variables as such. We're going to use unboxed tuples in place of the tuples we saw in `runState`, because it would be pretty terrible if we had to do heap allocation every time we performed an IO action.

The next primitive is `State# RealWorld`, which will correspond to the `s` parameter of our state monad. Actually, it's two primitives, the type constructor `State#`, and the magic type `RealWorld` (which doesn't have a `#` suffix, fascinatingly enough.) The reason why this is divided into a type constructor and a type parameter is because the `ST` monad also reuses this framework—but that's a matter for another blog post. You can treat `State# RealWorld` as a type that represents a very magical value: the value of the entire real world. When you ran a state monad, you could initialize the state with any value you cooked up, but only the `main` function receives a real world, and it then gets threaded along any IO code you may end up having executing.

One question you may ask is, "What about `unsafePerformIO`?" In particular, since it may show up in any pure computation, where the real world may not necessarily available, how can we fake up a copy of the real world to do the equivalent of a nested `runState`? In these cases, we have one final primitive, `realWorld# :: State# RealWorld`, which allows you to grab a reference to the real world wherever you may be. But since this is not hooked up to `main`, you get absolutely *no* ordering guarantees.

Hello World

Let's return to the Hello World program that I promised to explain:

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

When we compile this, we get some core that looks like this (certain bits, most notably the casts (which, while a fascinating demonstration of how newtypes work, have no runtime effect), pruned for your viewing pleasure):

```
Main.main2 :: [GHC.Types.Char]
Main.main2 = GHC.Base.unpackCString# "world!"

Main.main3 :: [GHC.Types.Char]
Main.main3 = GHC.Base.unpackCString# "Hello "

Main.main1 :: GHC.Prim.State# GHC.Prim.RealWorld
              -> (# GHC.Prim.State# GHC.Prim.RealWorld, () #)
Main.main1 =
  \ (eta_ag6 :: GHC.Prim.State# GHC.Prim.RealWorld) ->
    case GHC.IO.Handle.Text.hPutStr1
           GHC.IO.Handle.FD.stdout Main.main3 eta_ag6
    of _ { (# new_s_alV, _ #) ->
    case GHC.IO.Handle.Text.hPutStr1
           GHC.IO.Handle.FD.stdout Main.main2 new_s_alV
    of _ { (# new_s1_alJ, _ #) ->
    GHC.IO.Handle.Text.hPutChar1
      GHC.IO.Handle.FD.stdout System.IO.hPrint2 new_s1_alJ
    }
    }

Main.main4 :: GHC.Prim.State# GHC.Prim.RealWorld
              -> (# GHC.Prim.State# GHC.Prim.RealWorld, () #)
Main.main4 =
  GHC.TopHandler.runMainIO1 @ () Main.main1

:Main.main :: GHC.Types.IO ()
:Main.main =
  Main.main4
```

The important bit is `Main.main1`. Reformatted and renamed, it looks just like our desugared state monad:

```
Main.main1 =
  \ (s :: State# RealWorld) ->
    case hPutStr1 stdout main3 s  of _ { (# s', _ #) ->
    case hPutStr1 stdout main2 s' of _ { (# s'', _ #) ->
    hPutChar1 stdout hPrint2 s''
    }}
```

The monads are all gone, and `hPutStr1 stdout main3 s`, while ostensibly always returning a value of type `(# State# RealWorld, () #)`, has side-effects. The repeated case-expressions, however, ensure our optimizer doesn't reorder the IO instructions (since that would have a very observable effect!)

For the curious, here are some other notable bits about the core output:

- Our `:main` function (with a colon in front) doesn't actually go straight to our code: it invokes a wrapper function `GHC.TopHandler.runMainIO` which does some initialization work like installing the top-level interrupt handler.
- `unpackCString#` has the type `Addr# -> [Char]`, so what it does it transforms a null-terminated C string into a traditional Haskell string. This is because we store strings as null-terminated C strings whenever possible. If a null byte or other nasty binary is embedded, we would use `unpackCStringUtf8#` instead.
- `putStr` and `putStrLn` are nowhere in sight. This is because I compiled with `-O`, so these function calls got inlined.

The importance of being ordered

To emphasize how important ordering is, consider what happens when you mix up `seq`, which is traditionally used with pure code and doesn't give any order constraints, and IO, for which ordering is very important. That is, consider Bug 5129. Simon Peyton Jones gives a great explanation, so I'm just going to highlight how seductive (and wrong) code that isn't ordered properly is. The code in question is `x `seq` return ()`. What does this compile to? The following core:

```
case x of _ {
  __DEFAULT -> \s :: State# RealWorld -> (# s, () #)
}
```

Notice that the `seq` compiles into a `case` statement (since case statements in Core are strict), and also notice that there is no involvement with the `s` parameter in this statement. Thus, if this snippet is included in a larger fragment, these statements may get optimized around. And in fact, this is exactly what happens in some cases, as Simon describes. Moral of the story? Don't write `x `seq` return ()` (indeed, I think there are some instances of this idiom in some of the base libraries that need to get fixed.) The new world order is a new primop:

```
case seqS# x s of _ {
  s' -> (# s', () #)
}
```

Much better!

This also demonstrates why `seq x y` gives absolutely no guarantees about whether or not `x` or `y` will be evaluated first. The optimizer may notice that `y` always gives an exception, and since imprecise exceptions don't care which exception is thrown, it may just throw out any reference to `x`. Egads!

Further reading

- Most of the code that defines IO lives in the `GHC` supermodule in `base`, though the actual IO type is in `ghc-prim`. `GHC.Base` and `GHC.IO` make for particularly good reading.
- Primops are described on the GHC Trac.
- The ST monad is also implemented in essentially the exact same way: the unsafe coercion functions just do some type shuffling, and don't actually change anything. You can read more about it in `GHC.ST`.

- May 4, 2011
- Haskell

17 Responses to "Unraveling the mystery of the IO monad"

1.  *Conal Elliott* says:
   May 4, 2011 at 12:20 pm

   What confuses me about this story is that State is a model of *purely sequential*, determinstic computation, while IO has two forms of nondetermistic concurrency: forkIO (concurrent threads) and interaction (program/world concurrency).

2.  *Johan Tibell* says:
   May 4, 2011 at 1:32 pm

   Excellent post. Could you give an example of a function in the base libraries that is using this broken idiom?

3.  *Edward Z. Yang* says:
   May 4, 2011 at 2:22 pm

   Conal: Well…

   ```
   forkIO :: IO () -> IO ThreadId
   forkIO action = IO $ \ s ->
      case (fork# action_plus s) of (# s1, tid #) -> (# s1, ThreadId tid #)
    where
     action_plus = catchException action childHandler
   ```

   So, it looks like "fork#" does the magic of ensuring action_plus is run in another thread. A sort of runState, I suppose.

   Johan: Actually, it's not totally clear yet: if you don't care about ordering (and just strictness), the idiom might be OK. We're discussing this…

4.  *Conal Elliott* says:
[May 4, 2011 at 3:47 pm]

Edward: Thanks for digging up the code for forkIO. Is fork# defined via the State-like model of IO? If so, how? And if not (and I don't think it can be, considering nondeterminism), then the State "model" isn't really a model, and this persistent talk of IO as being explainable via State is untrue.

Why do I care about this Haskell myth of IO as State? Because denotational models are extremely useful as a basis for precise understanding and therefore correctness, derivation and optimization. And as long as people believe there is a denotational basis for IO, I don't expect they'll look for a denotationally sound alternative.

"The greatest obstacle to discovering the shape of the earth, the continents, and the oceans was not ignorance but the illusion of knowledge." – Daniel J. Boorstin

More on this topic at http://conal.net/blog/posts/is-haskell-a-purely-functional-language/ .

5.  *Edward Z. Yang* says:
[May 4, 2011 at 3:52 pm]

Conal: I think what we have here is an incomplete and inaccurate, but *adequate* denotational model for IO. It's not that IO as state isn't completely wrong, but it seems to work most of the time, so we don't worry too much when something strange like `fork#` comes along. Well, the GHC developers do—it's very important for us to have our semantic models right (as the bug quoted shows.)

I do think the intuition of fork# just "causing" another thread of execution (with its own real world token), completely unattached to the current one, mostly works, because the RealWorld is inherently multithreaded and you're in for a nasty surprise if you expect no one to change the real world while you're executing. So I suspect this is a more fundamental issue you would have to address first.

6.  *Conal Elliott* says:
[May 4, 2011 at 5:19 pm]

Your descriptions of the purported IO model as "incomplete and

inaccurate, but *adequate* denotational model" and "seems to work most of the time" are what I expect in the setting of typical programming languages in which "reasoning" exhibits a lot of hand-waving and self-deception. In contrast, denotative programming (roughly, Haskell minus IO) excels at supporting precise, tractable & correct reasoning.

So I'm baffled here. For *what* is an "incomplete and inaccurate" model adequate? And of what value is an understanding that merely "seems to work" and only "most of the time"?

It's easy to generate examples where the bogus State model of IO really doesn't work, i.e., where it yields invalid conclusions. For instance, we'd be tempted to equate "writeIORef r 3 >> writeIORef r 4" to "writeIORef r 4", which is correct in the State model and incorrect for IO.

7.  *yachris* says:
May 4, 2011 at 6:03 pm

   Hey Edward,

   Thanks for the interesting series… way over my head :-) but cool, none the less.

   One thing: early on in this article, I think a word is missing at the end of this sentence: "…things we handwave away is how the IO monad."

8.  *Edward Z. Yang* says:
May 4, 2011 at 6:11 pm

   yachris: Fixed.

   Conal: Thinking about this question.

9.  *JCAB* says:
May 4, 2011 at 6:42 pm

   Conal: AFAIK, modeling IO as a state monad is… an implementation artifact. It can just as well be done with primitive actions and primitive combinators. I think the way GHC does it is just for convenience: having as little as possible (barring primitives that are there for performance) that's "special" in the runtime.

Another way to see this is that a value of type “IO a” just defines a sequence of actions, where the one action can define what the subsequent actions are (that’s what makes it a Monad). Multiple values of types “IO a” represent multiple, independent sequences of actions. “forkIO”s argument is not part of the sequence that the “forkIO” is in, it’s just an entirely new sequence. Proof of this is that it cannot directly define the actions that come after the “forkIO”. Similarly, the rest of the world at large is not part of any sequences in your program. It can be modeled as separate, concurrent sequences, which your program (the IO sequences you define) can choose to interact with or not. Yes, IO happens to be a Monad (State or no State), but the monadic threading happens only during a single sequence.

“seq” is a pure function. There’s nothing in there that’s inherently related to ordering, except for its behavior when the first parameter doesn’t terminate. “seq x y” is different from “const y x” only in this respect. If “y” terminates, then “seq x y” cannot yield a value until it determines that “x” also terminates. There’s nothing wrong with an optimizer that ultimately replaces “seq (1 + 2) 0” with “0” (thus failing to evaluate “x” after all), and there’s nothing wrong with an optimizer that replaces “seq undefined (error “!”)” with “error (“!”)”.

Leaving aside the fact that “evaluate” seems to be an IO action like any other whereas “seq” is pure, the semantical idea of “evaluate” clearly doesn’t match the idea of “seq”, hence the bug. In the case of “evaluate”, “evaluate undefined >> error (“!”)” is not supposed to be equivalent to “error (“!”)”. Therefore, using “seq” to implement “evaluate” sounds suspect, at a minimum.

Anyway, the semantincs of “seq”, what it does or doesn’t do, are hard to wrap one’s head around. “evaluate” is GHC internals, which I don’t really know, so use your grain of salt with appropriate discretion, and please anyone let me know if I misunderstand anything.

Edward, I have to say I’m enjoying your posts more than usual, lately. Thank you!

10.  *Conal Elliott* says:
[May 5, 2011 at 6:48 pm]

JCAB: I also understand the IO/State link to be part of an implementation trick, not a truthful model.

> Another way to see this is that a value of type “IO a” just defines a sequence of actions …

Which raises the question “What is an action?”. And sequence isn’t really adequate, considering forkIO, right? (Unless you consider forking to be just an action). I’m looking for explanations precise enough to support correct (and only correct) reasoning. There’s no shortage of informal, imprecise explanations.

11. *What a Functionally Perfect Day! | Windows Live space* says:
October 19, 2011 at 4:58 am

    […] the first time that I’ve really put all of that together. There’s some good discussion in this blog […]

12. *Need some introduction to IO Monads specifics | michpodolsky* says:
September 5, 2014 at 9:24 am

    […] Is it here: http://blog.ezyang.com/2011/05/unraveling-the-mystery-of-the-io-monad/ ? […]

13. *Haskell Profiling and Optimization | NP-Incompleteness* says:
October 8, 2014 at 9:36 am

    […] functionality. I didn’t have time to read, but I’ve bookmark those for future reading: Unraveling the mystery of the IO monad and Tracing the compilation of Hello […]

14.  *bitonic* says:
January 13, 2015 at 6:19 pm

    > case x of _ {
    > __DEFAULT -> \s :: State# RealWorld -> (# s, x #)
    > }

    should read

    > case x of _ {
    > __DEFAULT -> \s :: State# RealWorld -> (# s, () #)
    > }

15.  *Edward Z. Yang* says:
January 13, 2015 at 6:21 pm

    Right, thanks you, fixed!

16.  *jacinabox* says:
[June 8, 2015 at 12:39 pm]

You might look at the “poor man’s concurrency monad,” it’s a pretty accurate model.

17.  *jacinabox* says:
[July 31, 2015 at 2:52 pm]

Continuing, an adequate model for the IO monad is the state machine:

data St s f t = St s (f (s -> Either s t))

The type s represents the state of the computer (in other words, a bit vector representing computer memory). It doesn’t need to stand for the state of the world, whatever that might mean. The functor provides several ways to continue from the state, either producing a new state, or terminating with a result.

To produce the execution of the program, the environment will iteratively execute the step function on the current state, until a result is reached. It can easily do input by updating part of the state in between steps, as devices do by writing directly to hardware memory.

The state can be divided by use:

type State = (R, W, X)

So that R is read memory, W is write memory, and X is memory private from the environment. Two state machines are equivalent if the R state behaves the same under all manipulations of the W state, modulo stuttering steps.

It is OK for machines to poll on a multi-process system. The reason is that the environment can detect when a machine’s update step yields no change in the state. In that case, the machine is stuck, and the environment can suspend it until something happens that would cause it to become unstuck.

Consider a program that polls for input:

puts(“Press enter to continue”);
while (last_character() != 32);

The call to ‘last_character’ causes no change at all in the state, so if the

program doesn’t find the character it wants first time, there is no point in running it until the next time the result of last_character changes. So programs that poll can be interpreted in an efficient way.

Leave a Comment

Name (Optional):

Comment:

Post Comment

« Previous Post Next Post »