A pragmatic look at monads in Haskell

Andy Gimblett

Department of Computer Science
University of Wales Swansea

No Grownups, 2007.05.11

Introduction

- Monads are essential but have a reputation for difficulty.
- “Intro to Monads” is a cottage industry – here’s my take.
- Most intros quite technical; ‘what are monads *really*?’ (λ-calculus and/or category theory)

- This talk: pragmatic; at the low end!
  - Doesn’t assume much knowledge of Haskell.
  - Aim: motivate / explain in a very practical sense.
  - Some rules of thumb & Handy Stuff To Know.

Haskell is pure and that's A Good Thing™

- Haskell is a **pure** functional language.
- λ-calculus all the way, baby!
- *Referentially transparent* functions:
  - Same behaviour wherever/whenever function is called.
- No explicit state, and no *side-effects*.
  - Side-effect: affecting later computations other than just by returning a value.
- 'Von Neumann' languages mainly Get Things Done via side effects (modifying state).
- 'State' in a pure functional program must be passed among its functions explicitly.

Purity is sweet

- Purity is Really Nice in a number of ways.

- Order of evaluation is irrelevant. . .
- . . . which makes *lazy evaluation* 'easy'.
- Enables memoization of functions.
- Reasoning about programs gets easier: *compositionality*. (Apparently some people actually do this.)
- Elegance, consistency, one model. . .

- Problem: not everything works that way.

Some things we do are not pure

- Many interesting/useful cases *aren't* pure.
- `random :: Int -> Int` (fictional)
- The Big One: I/O.
  - Output of `readLine` differs for each call.
  - Order of execution is significant – consider output.
- ∃ many cases where RT doesn't apply or isn't a win.
  - Impure functions (as above).
  - Cases where order of execution **is** significant;
  - Cases where state really is important, must be maintained/modified.
  - *e.g.* I/O, concurrency, exceptions, parsing, . . .

What to do about the impure stuff?

- One way: include ‘impure’ language constructs directly.
  - *i.e.* some parts of language pure; some have side-effects.
- *e.g.* Lisp, ML, Erlang, . . .
- Reasonable and pragmatic – gets things done.
- A compromise; we lose good stuff from previous slide.
  - Haskell: what happens when you don’t compromise.

- Haskell remains completely pure, by being Very Clever.

Isn't 'pure side-effects' an oxymoron?

- The loophole:
  - A pure function can't perform side-effects.
  - *Can* construct a value *representing* a side-effecting computation.
  - Caller then applies that computation when convenient.
- Monads encapsulate this idiom very nicely.
- So (*e.g.*) a value of type `IO String`:
  - Represents a computation which does some I/O . . .
  - . . . (*i.e.* has side effects) . . .
  - . . . and returns a value of type `String` when performed.
- When does the side-effect actually happen?
  - When the result of the computation is required.

History: how to do I/O in Haskell? (late 1990s)

- ("A History of Haskell: being lazy with class", SPJ 2007)
- Q: how to do I/O *well* without sacrificing purity?
- Two competing (but equivalent, it turns out) approaches:
  - *Streams* and *continuations*.
  - In both cases: `main :: [Response] -> [Request]`
  - Somewhat messy/roundabout. Neither feels 'great'.
- Meanwhile, Moggi/Wadler bring **monads** into the frame.
  - Turn out to *really* help with structuring pure programs.
  - Especially good at abstracting state & side effects.
  - Among other things, lead to a 'nicer' I/O model.
- *n.b.:* monads **not** a 'language extension' / added feature.
  - Live in the library; pure Haskell.
  - (Some syntactic sugar handled by compiler, though.)

Example: some I/O

```
main :: IO ()
main = do putStrLn "Hi"
          n <- getLine
          putStrLn n
```

- `main`'s type: no inputs; output is of type `IO ()`
- (Returns a computation which) does some I/O which doesn't return anything.
  - `()` is the empty type – think `void` in C.
  - Side effects of computation: the input/output it does!
- Similarly, we have:

  ```
  putStrLn ::  String -> IO ()
  ```

  - Takes a string, (returns a computation which) does some I/O which doesn't return anything.

Example: some I/O (2)

```
main :: IO ()
main = do putStrLn "Hi"
          n <- getLine
          putStrLn n
```

- What about `getLine`? Reads a string from stdin.
- We'd expect: "no input; returns a string". Sure enough:
  `getLine :: IO String`
- No inputs; output is of type `IO String`
- `getLine` (returns a computation which) . . .
- . . . does some side effects and returns a `String`.
  - Side-effect: reading a string from stdin.
- We 'store' that `String` in `n`.
- Then `putStrLn n` to print it out again.

Example: some I/O (3)

```
main :: IO ()
main = do putStrLn "Hi"
          n <- getLine
          putStrLn n
```

- `do` – notation for chaining monadic operations.
- Order of execution is obviously important here.
- Plus we have something that looks a lot like state.
- Isn't this just an 'imperative part of the language'?
- No: `do` notation is syntactic sugar for Something Clever.
- Desugared version:

```
main :: IO ()
main = putStrLn "Hi" >> getLine >>= \n -> putStrLn n
```

Example: some I/O (4)

```
main = putStrLn "Hi" >> getLine >>= \n -> putStrLn n
```

- `>>=` and `>>` are just normal pure functions.
  - (Called ‘bind’ and ‘anonymous bind’ respectively.)
- We won’t look at their details (as promised!).
- A monad consists of:
  - A type constructor & functions `return` & `>>=`
  - . . . which must obey certain laws for it to be a monad.
- `\n -> putStrLn n` is a λ-abstraction.
- The point: it all boils down to pure Haskell.
- Ordering & state ‘just happen’ because of evaluation rules.
  - *e.g.* can’t print `n` until we know `n`.
  - *e.g.* scope of `n` is as in a normal λ-expression.

Big Idea: Something Close To The Truth

A monad represents a computation which can perform a side-effect

Computation doesn't occur until it needs to (see lazy evaluation).

We chain these computations, and their results, together.

∃ Super Handy Functions which work for all types of monads.

Big Idea: Less True, But A Useful Model

A monad is a container for something.

Usually, something produced as the result of a non-pure operation.

`en.wikibooks.org/wiki/Haskell/Monads`
nuclear waste metaphor.

The pragmatics

- Monads neatly encapsulate side-effects:
  - clean separation between pure and impure worlds.
  - type system tells you when in the impure world.
- Difficulty: how to cross that divide? *e.g.* how to:
  - mix pure and impure functions in a real program?
  - pass result of a monadic op to a pure function?
  - use result of a pure function in a monad?
  - move things between different monad types?
  - (*e.g.*) turn `[IO String]` into `IO [String]`.
- All part of learning to work with Haskell, it seems.
- Good news: a few ‘rules of thumb’ & helper functions are what you need.

How do we structure a Haskell program?

- `main`'s type is `IO ()`
- (Loosely) `main` does some I/O & returns `()`.
- So the entry point of a Haskell program is monadic.
- *Doesn't* mean every function must return `IO` monad!
- Many (most?) functions we write/use are, in fact, non-monadic.
  - (Or involve other monads.)
- How is this managed?
- Turns out not to be too bad...
- Easy to get out of the monadic world when we want.

Moving from the `IO` world to outside it

- Some functions you write *must* involve `IO` monad.
  - *e.g.* reading input from user; writing output to user.
  - *e.g.* file I/O, network I/O, random number seeding.
- Functions which call those & manipulate their data: also 'in' `IO` monad.
  - In particular, `main`.
- However, at some point (in a do block, say):
  - Get the `a` out of the `IO a` ...
  - ... and pass it 'down' to a non-monadic function.
- Generally: only 'in' `IO` monad 'near top' of program.
  - (Very loose generalisation, but essentially sound.)

Example: leaving the `IO` world...

```
main :: IO ()
main = do files <- getDirectoryContents "hello"
          print files
          let other = nodots files
          print other

nodots :: [String] -> [String]
nodots x = filter notdot x
    where notdot f = not ((f == ".") || (f == ".."))
```

```
*Foo> main
[".","..","foo"]
["foo"]
```

Example: leaving the `IO` world (2)

```
getDirectoryContents :: FilePath -> IO [String]
nodots :: [String] -> [String]
```

1. We take result from (monadic) `getDirectoryContents`
2. We 'unwrap it', 'storing' in `files`
3. We pass that to `nodots`, which is **non-monadic**
   - No `IO` in its type signature.
   - Just operates on a lists of strings.
   - Doesn't care if they came from an I/O operation or not.
   - Which is exactly as it should be!
4. (Then we store that result in `other` and print it.
   nb: `print` has return type `IO ()`, same as `main`)

Helpful syntactic rules of thumb for do blocks

Simple ‘rules of thumb’ which help inside `do` blocks. . .

(Examples follow)

1. To bind result of a monadic function to `x`, use `x <-`
2. To bind result of a pure function to `x`, use `let x =`
3. Otherwise, line must be ‘in same monad’ as block
   - In particular, last line must have right return type!
   - Often: use `return` to wrap `a` as `M a`.

Binding the result of a monadic function

```
main :: IO ()
main = do files <- getDirectoryContents "hello"
          print files
          let other = nodots files
          print other
```

- `getDirectoryContents` returns `IO [String]`
- We 'store' that `[String]` in `files`
  - More accurately: we bind it to the name `files`
- `<-` 'extracts' the `a` from `IO a` and binds it.
- Can't extract from *just any* monad: must be same type as we're in.
- *e.g.*, couldn't do this to a value of type `Maybe a`
  - Unless, of course, `do` block's return type was `Maybe a`

Binding the result of a non-monadic function

```
main :: IO ()
main = do files <- getDirectoryContents "hello"
          print files
          let other = nodots files
          print other
```

- `nodots` returns `[String]`
- We 'store' that `[String]` in `other`
  - More accurately: we bind it to the name `other`
- Nothing much more to say, really!
- We've taken a value, and bound it to a name for future reference.
- OK! That was easy!

Non-binding lines must be in same monad as do block

```
main :: IO ()
main = do files <- getDirectoryContents "hello"
          print files
          let other = nodots files
          print other
```

- `main`'s type is `IO ()`
- Non-binding expressions must return in same monad.
- *e.g.* `print :: String -> IO ()`
- *e.g.* `getLine :: IO String` would also be OK
  - If not binding, would just throw away the string result.
- Obvious if you desugar the `do` block and look at types of `>>=`, etc.

Often use `return` on final line to get right type

- Final line must have right return type.
  - *e.g.* final line in an `IO ()` block *can't* be `IO String`.
  - Above: final call, `print`, returns `IO ()`

```
run :: AParser st PROCESS
run = do asKey runS
         es <- event_set
         return (Run es)
```

(Example from HETS – `tinyurl.com/2agoau`)

- `return` 'packs' a value into the monad if necessary.
  - `Run es` is a `PROCESS` constructor.
  - (In this example, the monad is `AParser st`)
- `asKey runS` – return type is `AParser st Token`

Using map with monads

- Consider the function `listFilesR`
  - Lists a directory's contents, recursively.
  - `listFilesR :: String -> IO [String]`
  - I/O operation, returning a list of `String`s.
- I want to do this for *multiple* directories.
- General approach would be to use `map`:
  - `map f l` calls `f` on every element of `l`
  - `map :: (a -> b) -> [a] -> [b]`
  - So `map listFilesR dirs` returns `[IO [String]]`
- Would prefer to get `IO [[String]]`
- Usually want the monad 'on the outermest level'.
- Just easier to work with that way. :-)

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

- `mapM` – "Monadic map"
  - If `m` is a monad...
  - ...take a function of type `a -> m b`, and list of `a`...
  - ...and return `m [b]`
- This is more like what we want:
  - `listFilesR :: String -> IO [String]` is our `a -> m b`
  - So we get back an `IO [[String]]`
  - Does some I/O, returns a list of lists of `String`s.
  - One list per directory in `dirs`.
- That's nicer to work with than `[IO [String]]`
- *e.g.* can easily work with it in do block...

mapM demo

```
main :: IO ()
main = do let dirs = ["hello", "bye"]
          nested <- mapM listFilesR dirs
          print nested
```

```
*Foo> main
[["hello/foo"],["bye/bar","bye/boing/gribble"]]
```

- Next: what I *really* want is `IO [String]`
  - ie, I want to flatten that list:
- `["hello/foo","bye/bar","bye/boing/gribble"]`

Lifting `concat` to the monadic world

```
*Foo> concat [["a"],["b","c"]]
["a","b","c"]
```

- `concat :: [[a]] -> [a]` – flatten a list of lists.
- `concat (mapM listFilesR dirs)` fails
  - Input to `concat` must be `[[a]]` not `m [[a]]`.
- We need a 'monadic `concat`', like our 'monadic map'.
  - No such thing, alas. But all is not lost!
- `liftM :: Monad m => (a -> b) -> m a -> m b`
  - Lifts function `a -> b` to `m a -> m b`

Lifting `concat` to the monadic world (2)

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

So `liftM concat (mapM listFilesR dirs)` is right.

```
main :: IO ()
main = do let dirs = ["hello", "bye"]
          nested <- mapM listFilesR dirs
          flat <- liftM concat nested
          print flat
```

```
*Foo> main
["hello/foo","bye/bar","bye/boing/gribble"]
```

Other handy functions for crossing the line

- We've seen `mapM` and `liftM`
- Various other helpers in standard prelude & libraries
- `liftM2`, `liftM3`, `liftM4`, `liftM5`
  - Lifting functions of 2, 3, 4, 5 inputs.
  - Need more? No, you don't, or you're insane. ;-)
- `filterM`, `foldM` – monadic `filter` & `fold`
- That's all I've needed so far. Only doing easy stuff, though!
- Your friends: hoogle and channel `#haskell`

Conclusion

- Monads are a key part of the Haskell worldview
- Unavoidable, and first met, for doing I/O.
  - But I/O is not the whole story, eg `Maybe`, Parsec, ...
- State & side-effects treated purely.
- `do` blocks are a nice syntactic sugar
  - Helps to know 'rules' when working within them
- Various functions help us move between the monadic & non-monadic 'worlds'.
- A strong influence on how we structure Haskell programs.