Fun With Haskell: Effects, Purely

Nathaniel Wesley Filardo

January 13, 2012

Metadata

- Anybody have questions from last time?

More Intro Stuff
Another Word On Laziness

- The `Debug.Trace` module offers

```
trace :: String -> a -> a
```

- When evaluated, `trace` prints out its first argument and then returns the second.
- `trace` is unsafe.
  - It should only be used for debugging.
- Why use it at all, then?
  - Reveals the act of computation.
  - Try: `map (+1) [1,2,trace "Hi" 3]`

More Intro Stuff
Deriving Instances

- There's often an obvious way to ascribe to an instance.

```
data Count = None | One | Few | Many
```

- Haskell can often **derive** instances.

More Intro Stuff
Deriving Instances

- There's often an obvious way to ascribe to an instance.

```
data Count = None | One | Few | Many
```

- Haskell can often **derive** instances.
- Support for six classes in the Haskell 98 standard:

DerivingEx.hs

```
data Count = None | One | Few | Many
  deriving (Bounded, Enum, Eq, Ord, Read, Show)
```

More Intro Stuff
Records

- `data`-types as defined so far are nice,
- but maybe not always what we want.
    - Thanks to LYAHFGG [7] for the example.
- A person has a first and last name, age, height.

```
data Person = Person String String Int Float
```

- And accessor functions:

```
firstName :: Person -> String
firstName (Person n _ _ _) = n
```

- Who wants to write all of those?

More Intro Stuff
Records

- A person has a first and last name, age, height.

PersonRecord.hs

```
data Person = Person
            { firstName :: String
            , lastName  :: String
            , age       :: Int
            , height    :: Float
            }
    deriving (Eq, Ord, Show)
```

- Accessors for free:

```
*Main> :type age
age :: Person -> Int
```

More Intro Stuff
Records

- Old-style constructors still work:

```
*Main> Person "N" "F" 27 170
```

- Cooler: pattern matching by label:

PersonRecord.hs

```
canVote (Person {age = x}) = x >= 18
```

- Record "update" syntax (clunky):

PersonRecord.hs

```
birthday p = p {age = (age p) + 1}
```

More Intro Stuff
Type Aliases and Newtypes

- Type Aliases provide alternative names:

```
type String = [Char]
type AssocList k v = [(k,v)]
```

- (Originally) Exact substitutability.
  - Strings are `Eq`-able because lists of `Eq`-able things are `Eq`-able and `Char` is `Eq`-able.
  - GHC language extension TypeSynonymInstances allow non-default semantics; don't worry about it.

More Intro Stuff
Type Aliases and Newtypes

- Suppose I wanted a type that's mostly like Int [4]:

```
data MyInt = MyInt Int deriving (Eq, Show)
instance Ord MyInt where {- ... -}
```

- Works, mostly.
  - Technically: The existence of both `MyInt` $\perp$ and $\perp$ means that `MyInt` is not isomorphic to `Int`.
- Inefficient: boxed (again)!

More Intro Stuff
Type Aliases and Newtypes

- `newtype` directives intended to give "mostly isomorphic" types.
- Try instead:

```
newtype MyInt = MyInt Int
  deriving (Eq, Read, Show)
instance Ord MyInt where {- ... -}
```

- Works!
- Constructed and destructed like `data MyInt`.
- Efficient: MyInt box exists only at compile time.

More Intro Stuff
Type Aliases and Newtypes

- `newtype` directives intended to give "mostly isomorphic" types.
- Can have only one constructor, with exactly one argument.
- These don't work:

```
newtype NTBool = True | False
newtype NTSPair a = NTSPair a a
newtype NTPair a b = NTPair a b
```

Enter: Monads

Monads have been said to be ...

- Burritos
- Elephants
- "Just a monoid in the category of endofunctors, what's the problem?" [5] paraphrasing [6].
  - Also: "A monad is just a lax functor from a terminal bicategory, duh. fuck that monoid in category of endofunctors shit" [2]
- Trees With Grafting [3]

There are at least 35 known "monad tutorials" of various shapes and sizes; `http://www.haskell.org/haskellwiki/Monad_tutorials_timeline`.

Enter: Monads
Computations Which Might Abort

- Consider a set of functions

```
foo, bar, baz :: Int -> Int
```

- That we want to compose:

```
fbb x = baz (bar (foo x))

-- Shorter, "point-free" form:
fbb' = baz . bar . foo
```

Enter: Monads
Computations Which Might Abort

- Now consider a set of "failable" functions

```
foo, bar, baz :: Int -> Maybe Int
```

- Challenge: compose these.

```
fbb x = case foo x of
  Nothing -> Nothing
  Just fx -> case bar fx of
    Nothing -> {- aaaaa! -}
```

- There's gotta be a better way.
- A first example of "the monad pattern."

Enter: Monads
Computations Which Might Abort

- Consider a set of functions

```
foo, bar, baz :: Int -> Maybe Int
```

- Insight: potential successes combine.
    - Like case analysis above!
- Want a combinator

```
bindMaybe Nothing  _ = Nothing
bindMaybe (Just a) f = f a
```

Enter: Monads
Computations Which Might Abort

- Consider a set of functions

```
foo, bar, baz :: Int -> Maybe Int
```

- Insight: potential successes combine.
    - Like case analysis above!
- Want a combinator

```
bindMaybe Nothing  _ = Nothing
bindMaybe (Just a) f = f a
```

- Type is going to become familiar:

```
bindMaybe :: Maybe a -> (a -> Maybe b)
          -> Maybe b
```

Enter: Monads
Computations Which Might Abort

- Given

```
foo, bar, baz :: Int -> Maybe Int

bindMaybe :: Maybe a -> (a -> Maybe b)
          -> Maybe b
bindMaybe Nothing  _ = Nothing
bindMaybe (Just a) f = f a
```

- Now

```
fbb x = (foo x) `bindMaybe` bar
                `bindMaybe` baz
```

Enter: Monads
Computations Which Might Abort

- Is anybody else bothered by this?

```
fbb x = (foo x) ‘bindMaybe‘ bar
                ‘bindMaybe‘ baz
```

Enter: Monads
Computations Which Might Abort

- Is anybody else bothered by this?

```
fbb x = (foo x) `bindMaybe` bar
                `bindMaybe` baz
```

- Why is `foo` so different?

Enter: Monads
Computations Which Might Abort

- Is anybody else bothered by this?

```
fbb x = (foo x) `bindMaybe` bar
                `bindMaybe` baz
```

- Why is `foo` so different?
- Would rather have uniformity in steps.

```
fbb x = (Just x)
        `bindMaybe` foo
        `bindMaybe` bar
        `bindMaybe` baz
```

Enter: Monads
Environments

- Sometimes, we want to have read-only state available to us.
  - e.g. command line arguments
- Say that code is running in an **environment**.
- If `f :: a -> b` needs access to environment, make it `f :: e -> a -> b` or `f :: a -> e -> b`.

Enter: Monads
Environments

- What if we realize that our functions need access to the environment?

```
type Env = -- ...
foo, bar, baz :: Int -> Env -> Int
fbb = -- ... ?
```

- Challenge: compose them!

Enter: Monads
Environments

- Have made

```
foo, bar, baz :: Int -> Env -> Int
```

- Composing:

```
fbb x e = baz (bar (foo x e) e) e
```

- Still not so much fun, is it?

Enter: Monads
Environments

- Have made

```
foo, bar, baz :: Int -> Env -> Int
```

- Insight: `Env -> ...` all handled the same.
    - Fed same environment to each one.
- Define an alias

```
newtype Reader e a = Reader
                     { runReader :: e -> a }
```

- Now need to compose readers together.

Enter: Monads
Environments

- Now need to compose readers together.

Enter: Monads
Environments

- Now need to compose readers together.
- That is, we want something like

```
bindReader :: Reader e a -> (a -> Reader e b)
           -> Reader e b
```

- Look familiar?

Enter: Monads
Environments

- Defined

```
newtype Reader e a = Reader
                   { runReader :: e -> a }
bindReader :: Reader e a -> (a -> Reader e b)
           -> Reader e b
```

- Read off the types to guide implementation:

```
bindReader (Reader a) f =
  Reader (\e -> (runReader (f (a e))) e)
```

Enter: Monads
Environments

- Defined

```
newtype Reader e a = Reader
                   { runReader :: e -> a }
bindReader :: Reader e a -> (a -> Reader e b)
           -> Reader e b
```

- Now compose:

```
fbb :: Int -> Reader Env Int
fbb x = (foo x) `bindReader` bar `bindReader` baz
```

- Or, for uniformity:

```
fbb x = (Reader (const x)) `bindReader` foo
          `bindReader` bar `bindReader` baz
```

Enter: Monads
Keeping Counts

- Let's say we've defined

ExpensiveFib.hs

```
fib 0 = 1
fib 1 = 1
fib n | n >= 2 = fib (n-1) + fib (n-2)
fib _ = error "negative fib"
```

- And we want to measure just how many calls are made
    - One way: thread a counter through.
    - (May be better ways we can talk about later)
- Need to change the type:

```
type Ctr = Int
fibCtr :: Int -> Ctr -> (Int, Ctr)
```

Enter: Monads
Keeping Counts

- Also need a new implementation:

```
fibCtr :: Int -> Ctr -> (Int, Ctr)
fibCtr 0 c = (1, c+1)
fibCtr 1 c = (1, c+1)
fibCtr n c = -- ...
```

- Hm. Clearly, I need to call `fibCtr` on `(n-1)` and `(n-2)`.
- What do I do about the counter?

Enter: Monads
Keeping Counts

- Also need a new implementation:

```
fibCtr :: Int -> Ctr -> (Int, Ctr)
fibCtr 0 c = (1, c+1)
fibCtr 1 c = (1, c+1)
fibCtr n c = -- ...
```

- Hm. Clearly, I need to call `fibCtr` on `(n-1)` and `(n-2)`.
- What do I do about the counter?
- This mess:

```
fibCtr n c = let
    (a, c') = fibCtr (n-1) c
    (b, c'') = fibCtr (n-2) c'
 in (a+b, c'' + 1).
```

Enter: Monads
Keeping Counts

- Yuck!

```
fibCtr 0 c = (1, c+1)
fibCtr 1 c = (1, c+1)
fibCtr n c = let
    (a, c') = fibCtr (n-1) c
    (b, c'') = fibCtr (n-2) c'
 in (a+b, c'' + 1).
```

- Insight: State is like an environment where the *previous functions* get a chance to change it.

Enter: Monads
Keeping Counts

- Insight: State is like an environment where the *previous functions* get a chance to change it.
- So:

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

get :: State s s
get = State (\s -> (s,s))

put :: s -> State s ()
put s = State (\_ -> ((),s))
```

Enter: Monads
Keeping Counts

- Insight: State is like an environment where the *previous functions* get a chance to change it.
- So:

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

get :: State s s
get = State (\s -> (s,s))

put :: s -> State s ()
put s = State (\_ -> ((),s))
```

- What's the other thing we want?

Enter: Monads
Keeping Counts

- Insight: State is like an environment where the *previous functions* get a chance to change it.
- A State bind combinator:

```
bindState :: State s a -> (a -> State s b)
          -> State s b
```

Enter: Monads
Keeping Counts

- Insight: State is like an environment where the *previous functions* get a chance to change it.
- A State bind combinator:

```
bindState :: State s a -> (a -> State s b)
          -> State s b
```

- Sure:

```
bindState sa f = State (\s -> let
  (a, s’) = runState sa s in
  runState (f a) s’)
```

Enter: Monads
Keeping Counts

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

- Thusly armed, define a utility function:

```
constState :: a -> State s a
constState x = State (\s -> (x, s))
```

Enter: Monads
Keeping Counts

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

- Thusly armed, define a utility function:

```
constState :: a -> State s a
constState x = State (\s -> (x, s))
```

- And now a trickier one:

```
modify :: (s -> s) -> State s ()
modify f = get
  `bindState` (\s -> constState (f s))
  `bindState` put
```

Enter: Monads
Keeping Counts

Now revisit `fibCtr`.

- Base cases:

```
fibSCtr 0 = modify (+1)
`bindState` \() -> constState 1
```

- "First, adjust the counter by +1."
- "*Then*, ignore the counter and return 1."
- Haskell is a funny dialect of English: "and then" is pronounced "bind."

Enter: Monads
Keeping Counts

Now revisit `fibCtr`.

- Inductive case:

```
fibSCtr n = modify (+1)
  `bindState` \() -> fibSCtr (n-1)
  `bindState` \a  -> fibSCtr (n-2)
  `bindState` \b  -> constState (a+b)
```

- "First, adjust the counter by +1."
- "*Then*, call `fibSCtr (n-1)` and call the result `a`."
- "*Then*, call `fibSCtr (n-2)` and call the result `b`."
- "Then, ignore the counter and return `a+b`."

Enter: Monads
Keeping Counts

Now revisit `fibCtr`.

- Inductive case:

```
fibSCtr n = modify (+1)
  `bindState` \() -> fibSCtr (n-1)
  `bindState` \a  -> fibSCtr (n-2)
  `bindState` \b  -> constState (a+b)
```

- "First, adjust the counter by +1."
- "*Then*, call `fibSCtr (n-1)` and call the result `a`."
- "*Then*, call `fibSCtr (n-2)` and call the result `b`."
- "Then, ignore the counter and return `a+b`."
- (Don't worry, idiomatic Haskell is much cleaner. We'll get there.)

Enter: Monads
Keeping Counts

So, now we have:

```
fibSCtr :: Int -> State Int Int
fibSCtr 0 = modify (+1)
`bindState` \() -> constState 1
fibSCtr 1 = modify (+1)
`bindState` \() -> constState 1
fibSCtr n = modify (+1)
  `bindState` \() -> fibSCtr (n-1)
  `bindState` \a  -> fibSCtr (n-2)
  `bindState` \b  -> constState (a+b)
```

And we can actually run the thing with

```
*Main> runState (fibSCtr 20) 0
(10946,21891)
```

Enter: Monads
Monads For Real

Everybody ready for the real definition of monads?

Enter: Monads
Monads For Real

- A monad is
  - an endofunctor $T : C \to C$ with
  - a natural transformation $\eta : 1_C \to T$ and
  - a natural transformation $\mu : T^2 \to T$
  - such that

$$\begin{tikzcd} T^3 \arrow[r, "T\mu"] \arrow[d, "\mu T"] & T^2 \arrow[d, "\mu"] \\ T^2 \arrow[r, "\mu"] & T \end{tikzcd} \qquad \begin{tikzcd} T \arrow[r, "\eta T"] \arrow[d, "T\eta"] & T^2 \arrow[d, "\mu"] \\ T^2 \arrow[r, "\mu"] & T \end{tikzcd}$$

Enter: Monads
Monads For Real

- A monad is
  - an endofunctor $T : C \to C$ with
  - a natural transformation $\eta : 1_C \to T$ and
  - a natural transformation $\mu : T^2 \to T$
  - such that

$$\begin{tikzcd} T^3 \arrow[r, "T\mu"] \arrow[d, "\mu T"] & T^2 \arrow[d, "\mu"] \\ T^2 \arrow[r, "\mu"] & T \end{tikzcd} \qquad \begin{tikzcd} T \arrow[r, "\eta T"] \arrow[d, "T\eta"] & T^2 \arrow[d, "\mu"] \\ T^2 \arrow[r, "\mu"] & T \end{tikzcd}$$

- Uh. . . can I call a friend?

Enter: Monads
Monads For Real

- Let's try that again. A Monad is a type class:

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

Enter: Monads
Monads For Real

- Let's try that again. A Monad is a type class:

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

- Monad instances should obey
  - Left and right identity:

```
return a >>= f      === f a
m >>= return        === m
```

  - Associativity:

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

Enter: Monads
Monads For Real

- Maybe is a Monad:

```
instance Monad Maybe where
    return = Just

    Nothing  >>= _ = Nothing    -- bindMaybe
    (Just a) >>= f = f a
```

- Check the identity laws:

```
Just a   >>= f    === f a
Nothing  >>= Just === Nothing
(Just a) >>= Just === Just a
```

- Associativity similarly easy to check.

Enter: Monads
Monads For Real

- `Reader e` is a Monad:

```
instance Monad (Reader e) where
  return x = Reader (const x)
  ra >>= f =
    Reader (\e -> (runReader (f (a e))) e)
```

Enter: Monads
Monads For Real

- `Reader e` is a Monad:

```
instance Monad (Reader e) where
  return x = Reader (const x)
  ra >>= f =
    Reader (\e -> (runReader (f (a e))) e)
```

- `State s` is a Monad:

```
instance Monad (State s) where
  return x = State (\s -> (x,s))
  sa >>= f = State (\s ->
    let (a, s') = runState sa s
     in runState (f a) s')
```

Enter: Monads
Monads For Real

- >> is like >>= but ignores the result.
  - First computation run entirely for *effects*.

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

- Revisiting `fibSCtr`:

```
fibSCtr 0 = alter (+1) >> return 1
fibSCtr 1 = alter (+1) >> return 1
fibSCtr n = alter (+1)
   >>        fibSCtr (n-1)
   >>= \a -> fibSCtr (n-2)
   >>= \b -> return (a+b)
```

- Still sort of ugly, right?

Enter: Monads
Do Notation

- Haskell provides the wonderful and amazing *do* notation
  - Sometimes called “reprogrammable semicolon”
- Let’s try that again:

```
fibSCtr 0 = do
  alter (+1)
  return 1
```

Enter: Monads
Do Notation

- Haskell provides the wonderful and amazing *do* notation
  - Sometimes called “reprogrammable semicolon”
- Let’s try that again:

```
fibSCtr 0 = do
  alter (+1)
  return 1
```

- And the induction step?

```
fibSCtr n = do
  alter (+1)
  a <- fibSCtr (n-1)
  b <- fibSCtr (n-2)
  return (a+b)
```

Monads for Effect
What, exactly, are effects?

- Anything which depends on...
  - The Real World.
  - The order of execution.
- Things like
  - Ordered state
  - Mutable references
  - I/O: (Files, User, Network, Time, Random numbers, ...)
  - Catching exceptions

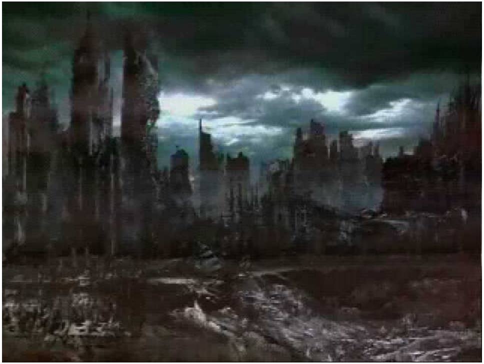
Monads for Effect
A Historical Parenthetical

- Haskell originally used *lists* for I/O:
  - Programs given an infinite list of input events
  - Programs produced a list of output events
- "The User" is a (particularly slow) thunk.
- Sort of worked, but extremely unpleasant.
  - Not a crazy idea in all cases.
  - Infinite, lazy list of random numbers?

Monads for Effect
IO Monad

- "The One-stop Sin Bin"
- Contains all sorts of goodies:
    - Mutable references
    - Multiple threads and thread-safe mutable references
    - `StableNames`,
    - Exception catching,
    - Files, Sockets, X11,
    - ...

Monads for Effect
IO Monad



(With apologies to The Matrix, `http://matrix.wikia.com`)

Monads for Effect
IO Monad

- OK, it's not so bad as all that.
- Functions which do IO can
    - interrogate the real world
    - make changes to the real world

Monads for Effect
IO Monad

- OK, it's not so bad as all that.
- Functions which do IO can
  - interrogate the real world
  - make changes to the real world
- IO is (essentially) State RealWorld.
  - Without `get` and `put`.
  - With other functions instead.

Monads for Effect
Revisiting Hello World

- Remember this?

HelloWorld.hs

```
main = putStrLn "Hello, World"
```

- Well

```
*Main> :type main
main :: IO ()
```

- Change to real world: “Hello, World!” now on screen.

Monads for Effect
Revisiting Hello World

- No *safe* way to "run IO and get the result" in pure code.
    - With good reason!
    - I/O can see the order of execution.
    - Lazy, pure code is supposed to be *independent* of evaluation order!
    - (We can talk about "benign effects" later.)
- Type of entire Haskell program is `IO ()`:
    - An I/O computation being run *entirely for its effects*.

Next time

- More on Monads and Effects
- More on I/O in particular
  - Programming with IO actions.
  - Brain teaser for next time:

```
twice a = a >> a
main = twice (putStrLn "Hello, World")
```

- Monads Atop Monads ("Monad Transformers")

Bib

- Available from: `http://courses.cms.caltech.edu/cs11/material/haskell/index.html`.
- Haskell weekly news: Issue 149.
  Available from: `http://www.haskell.org/pipermail/haskell-cafe/2010-February/072986.html`.
- Monads are trees with grafting, 2010.
  Available from: `http://blog.sigfpe.com/2010/01/monads-are-trees-with-grafting.html`.
- Hal Daumé III.
  Yet another haskell tutorial.
  2002–2006.
  Available from: `http://www.cs.utah.edu/~hal/htut/`.

James Iry.
A brief, incomplete, and mostly wrong history of programming languages, May 2009.
Available from: http://james-iry.blogspot.com/2009/05/brief-incomplete-and-mostly-wrong.html.

Saunders Mac Lane.
Categories for the Working Mathematician.
Springer, 1998.

Mirian Lipovača.
Learn You A Haskell For Great Good!
No Starch Press, April 2011.
Available from: http://learnyouahaskell.com/.