Lecture 4: Monads!

CSC 131
Spring, 2019

Kim Bruce

Homework

- Turn in using submit.cs.pomona.edu
- For second homework, turn in two files:
  - pdf file with complete homework solutions (including Haskell code)
  - file with hs suffix that contains only executable programs (so we can test your code)
  - put in folder and zip them up to submit.

Start Simpler: Functor

- Modeled on map function on lists

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b

instance Functor ([]) where
  fmap = map
```

  - [] here means operator that takes a type and makes it into a list type
- See later how can use with trees or other structured data

Trees are functors too!

```
data Tree a = Niltree | Maketree (a, Tree a, Tree a)
                deriving Show

instance Functor Tree where
  fmap f Niltree = Niltree
  fmap f (Maketree (root, left, right)) =
        Maketree (f root, fmap f left, fmap f right)
```

Functor Laws

- fmap id = id   -- 1st functor law
- fmap (g . f) = fmap g . fmap f -- 2nd functor law
  - "." is function composition
  - Can write fmap as an infix operator with <$>
    - Thus fmap f elts = f <$> elts
    - Makes it look more like function application

Maybe

- *data* Maybe a = Nothing | Just a *deriving* (Eq, Show)
  - Useful for computations that may not have a result
  - Part of "standard prelude" imported by all Haskell modules
  - Look up a phone number for a person.
  - Maybe Integer includes Nothing, Just 7, ...

Maybe is a Functor

- Sometimes may not get an answer:
  - E.g, look up something that may not be there.

```
data Maybe a = Just a | Nothing
    deriving (Eq, Ord)

instance Functor Maybe where
  fmap f Nothing = Nothing
  fmap f (Just v) = Just (f v)
```

Function f slides under "Just"
so negate <$> (Just 3) == fmap negate (Just 3)
== Just (-3)

Applying ourselves!

- What about binary functions like add?
- Can use Applicative Functor

```
class (Functor f) => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b

instance Applicative Maybe where
  pure         = Just
  (Just f) <*> (Just x) = Just (f x)
  _    <*> _    = Nothing
```

Binary Operators

- *Want* (+) … Just 2 … Just 3 == Just (2 + 3)
  - *Note* (+) <$> Just 2 == Just (2+)
    - (2+) *is same as* \x -> 2 + x
  - *Recall* (Just f) <*> Just x == Just (f x)
  - *Now* (+) <$> Just 2 <*> Just 3 == Just (2+) <*> (Just 3)
    == Just (2 + 3) == Just 5
- Pure wraps value with "Just", while <*> allows function application under "Just"

Summary

- Applicative has fmap or <$> from Functor
  - <$> allows function to apply under Just
  - adds pure, which wraps value with "Just"
  - <*> to allow Just function to apply to Just value
- Rules:
  - pure id <*> v = v -- Identity
  - pure f <*> pure x = pure (f x) -- Homomorphism
  - …

Let's get more complicated!

Using Maybe as Monad

- dormRooms =[("Jack",10),("Jill",20),("Ann",20)]
- phonesForRooms = [(10,23434),(20,23435),(30,23438)]
- getDormFor name [] = Nothing
  -- *2nd arg is name-room pairs*
  getDormFor name ((nm,rm):rest) = if nm == name
  then Just rm
  else getDormFor name rest
- getPhoneForRoom rm [] = Nothing
  getPhoneForRoom rm ((rmnum,phone):rest) =
  if rm == rmnum then Just phone
  else getPhoneForRoom rm rest

Awkward to Compose

- getPhoneForName name rooms phones =
  case getDormFor name rooms of
  Nothing -> Nothing
  Just rm -> getPhoneForRoom rm phones

- Must unwrap values to use and then rewrap
  - Applicative won't work!
- Easier if could write:
  - getPFN name rooms phones =
    do rm <- getDormFor name rooms
    num <- getPhoneForRoom rm phones
    return num
  - and not have to worry about error cases!

Defining Monads

part of Standard Prelude

- class Applicative m => Monad m where
  (>>=) :: m a → (a → m b) → m b
  return :: a → m a
  fail:: string → m a
  - >>= allows a kind of composition of wrapped values or computations -- called *bind*
  - return wraps an unwrapped value.
  - fail takes error string & aborts program
  - a >> b *abbreviates* a >> _ -> b *(constant fcn)*

Maybe Monad

- instance Monad Maybe where *part of Standard Prelude*
  (>>=) Nothing f = Nothing
  (>>=) (Just x) f = f x
  return x = Just x
  fail s = Nothing
- >>= preserves "Nothing",
- >>= unwraps argument to compute w/ a Just'ed value
- Second arg of >>= is function applied to unwrapped value
- *Abbreviate* compu >>= \x → exp *as*
  do x <- compu
  exp

Back to Example

- Expression
  - getPFN name rooms phones =
    do rm <- getDormFor name rooms
    num <- getPhoneForRoom rm phones
    return num
  - *abbreviates*
  - getPFN name rooms phones =
    getDormFor name rooms >>=
    (\rm -> getPhoneForRoom rm phones)

Monads

- Provide operations to compose wrapped values
- Operations obey laws:
  - return x >>= f == f x *left identity*
  - c >>= return == c *right identity*
  - c >>= (\x -> f x >>= g) == (c >>= f) >>= g *associativity*

In "do" notation

- Left identity:

```
do { x' <- return x;
     f x'              ≡    do { f x }
   }
```

- Right identity:

```
do { x <- m;
     return x          ≡    do { m }
   }
```

- Associativity:

```
do { y <- do { x <- m;            do { x <- m;
               f x                     do { y <- f x;
             }                 ≡                g y
     g y                                   }
}                                 }
```

Application of Laws

- Program:

```
skip_and_get = do
    unused <- getLine
    line <- getLine
    return line
```

- is equivalent to:

```
skip_and_get = do
    unused <- getLine
    getLine
```

by right identity

See http://www.haskell.org/haskellwiki/Monad_laws for more info

Other Monad Examples

- Error handling $M(a) = a \cup \{error\}$
  - Add a special "error value" to a type
  - Define bind operator ">>=" to propagate error
- Information-flow tracking $M(a) = a \times Labels$
  - Add information flow label to each value
  - Define bind to check and propagate labels
- State $M(a) = a \times States$
  - Computation produces value and new state
  - Define bind to make output state of first go to input state of second

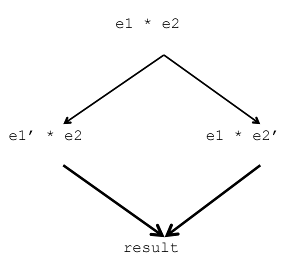
Big Idea

- Write code as though computing on a, but actually run it on M a.
  - That's what we did with Maybe monad!

Beauty

- Functional programming is beautiful:
  - Concise and powerful abstractions
    - higher-order functions, algebraic data types, parametric polymorphism, principled overloading, ...
  - Close correspondence with mathematics
    - Semantics of a code function is the mathematical function
    - Equational reasoning: if x = y, then f x = f y
  - Independence of order-of-evaluation
    - Confluence, aka Church-Rosser

Confluence means ...



- The compiler can choose the best sequential or parallel evaluation order!

... and the Beast

- But to be useful as well as beautiful, a language must manage the "Awkward Squad":
  - Input/Output
  - Imperative update
  - Error recovery (eg, timeout, divide by zero, etc.)
  - Foreign-language interfaces
  - Concurrency control

The whole point of a running a program is to interact with the external environment and affect it

The Direct Approach

- Just add imperative constructs "the usual way"
  - I/O via "functions" with side effects:
    - putChar 'x' + putChar 'y'
  - Imperative operations via assignable reference cells:
    - z = ref 0; z := z + 1; ...
  - Error recovery via exceptions
  - Foreign language procedures mapped to "functions"
  - Concurrency via operating system threads
- Can work if language determines eval order

  Examples: ML, OCAML, Scheme/Racket

What if Lazy?

- Order of evaluation deliberately undefined.
- Example:
  - ls = [putChar 'x', putChar 'y']
  - if only use (length ls), then nothing printed!!

Fundamental Question

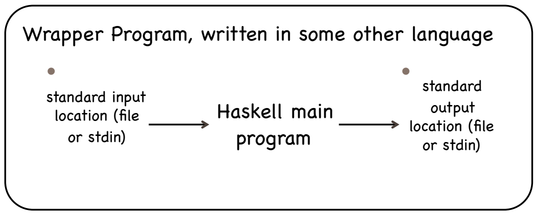
- Can you add imperative features with changing the meaning of pure Haskell expressions?
  - Even though laziness and side-effects are incompatible!!

History

- Big embarrassment to lazy functional programming community
  - ML, Scheme/LISP/Racket didn't care about being purely functional
- Alternatives:
  - Streams ⟵ *Haskell 1.0 adopted, essentially lazy lists*
  - Continuations
    - pure functions passed to IO routines to process input
  - Pass state of world as parameter
    - Hard to make single-threaded

Stream Model

- Move side effects outside of functional program
  - main:: String -> String



  - But what if more than one file or socket or ...?

Stream Model

- Enrich argument and return type of main to include all input and output events.

```
main :: [Response] -> [Request]
data Request   =  ReadFile Filename
          |  WriteFile FileName String
          | …
data Response  =  RequestFailed
          |  ReadOK String
          |  WriteOk
          |  Success  | …
```

- Wrapper program interprets requests and adds responses to input.

Stream Model is Awkward!

- Hard to extend
  - New I/O operations require adding new constructors to Request and Response types, modifying wrapper
- Does not associate Request with Response
  - easy to get "out-of-step," which can lead to deadlock
- Not composable
  - no easy way to combine two "main" programs
- ... and other problems!!!

Monads to Rescue!

- Value of type (IO a) is an action
  - that may perform some input/output
  - and deliver result of type a

I/O

- main :: IO( )   -- "IO action"
- main = putStrLn "Hello World!"
- where putStrLn:: String → IO( )
- getLine :: IO String   -- "IO action" returning string

• Want echo = putStrLn getLine
  - Types don't match
  - Need >> = for IO monad!!
  - echo = do str <- getLine
              putStrLn str

See monad.hs

Connecting Actions

getLine   IO String

String   *putStrLn*   IO ( )

Glued together with >>=

More IO

```
ask :: String -> String -> IO()
ask prompt ansPrefix =
        do putStr (prompt++" ")
           response <- getLine
           putStrLn (ansPrefix ++ " " ++ response)

getInteger :: IO Integer
getInteger = do putStr "Enter an integer: "
                line <- getLine
                return (read line)
        -- converts string to Integer then to IO Integer
```

IO & Ref Transparency

- Main program is IO action w/type IO( )
- Perform IO in IO actions & call pure functions from inside there
- Can never escape from IO! *Unlike Maybe.*
  - *No constructors for IO, so can't pattern match to escape!!!*
- IO impure in that successive calls of getLine return different values.

Using IO in Haskell

- Can build language at IO monad level:

```
ifIO :: IO Bool -> IO a -> IO a -> IO a
ifIO b tv fv = do { bv <- b;
                    if bv then tv else fv}

whileIO :: IO Bool -> IO() -> IO()
whileIO b m = ifIO b
                (do {m; whileIO b m})
                (return())
```