Programming Skills: Functional Programming

More about Monads

Examples from Hutton and Newbern

and Wikibooks Haskell

http://www.haskell.org/all_about_monads/html/index.html
http://en.wikibooks.org/wiki/Haskell
Definition and Axioms

\[
\begin{align*}
\text{class Monad } & m \text{ where} \\
\land (>>=) \land : & m \ a \to (a \to m \ b) \to m \ b \\
\land (>>) \land : & m \ a \to m \ b \to m \ b \\
\land \text{return} \land : & a \to m \ a \\
\land \text{fail} \land : & \text{String} \to m \ a \\
\land m \gg k \land = & m \gg= \_ \to k \\
\land \text{fail } s \land = & \text{error } s
\end{align*}
\]

right unit  \quad m \gg= \text{return} \quad = \quad m

left unit  \quad \text{return } x \gg= f \quad = \quad f \ x

associativity  \quad (m \gg= f) \gg= g \quad = \quad m \gg= (\_x \to f \ x \gg= g)
Id: trivial functions for do

eval (Leaf n) = return n

eval (Bin left opr right) =
  do l <- eval left
     r <- eval right
     return (op opr l r)
**Id: trivial functions for** `do`

```haskell
eval (Leaf n) = return n
eval (Bin left opr right) =
  do l <- eval left
     r <- eval right
     return (op opr l r)
```

```haskell
eval (Leaf n) = return n
eval (Bin left opr right) = eval left >>= \ l ->
   eval right >>= \ r ->
   return (op opr l r)
```
**Id: trivial functions for do**

```haskell
eval (Leaf n) = return n

eval (Bin left opr right) =
  do l <- eval left
     r <- eval right
     return (op opr l r)
```

```haskell
data Id a = Id a
instance Monad Id where
  return x = Id x
  Id x >>= f = f x
```
Maybe

```haskell
data Maybe a = Nothing | Just a
instance Monad Maybe where
  return         = Just
  fail           = Nothing
  Nothing  >>= f = Nothing
  (Just x) >>= f = f x
```

- Computation which can fail.
- Check the axioms!
Example: `lookup`

```haskell
lookup :: Eq a => a  -- a key
       -> [(a, b)]   -- the lookup table to use
       -> Maybe b    -- the result of the lookup

getTaxOwed name = do
  number       <- lookup name phonebook
  registration <- lookup number governmentDatabase
  lookup registration taxDatabase
```

- Computation 'stops' as soon as a lookup fails.
- Types may change along the chain.
class can have only one type parameter.

instance can only use type variables.
Either

— data Either err a = Left err | Right a

instance Monad (Either err) where
  return a        = Right a
  Left err >>= _ = Left err
  Right a >>= f   = f a
Exception handling

- `throw x = Left x`
- `onErr (Left x) handler = handler x`
- `onErr (Right a) _ = Right a`
- `test n = do x <- Right 10
  y <- Right n
  if y /= 0 then return (x/y)
  else throw 123
  `onErr` \s -> Left $ "error: " ++ (show s)`

- **try/catch control structure.**
- **handler must return the same type.**
List

instance Monad [] where
  m >>= k = concat (map k m)
  return x = [x]
  fail s = []

- Maybe: computation which can fail.
- List: computation with multiple results.
Example: game expansion

move:: Board -> [Board]
-- maps given board to all next boards
move1:: [Board] -> [Board]
-- maps all boards to all next boards after 1 move
move1 boards = concat (map move boards)
move3:: Board -> [Board]
-- maps board to all boards after 3 moves
move3 board = move1 $ move1 $ move1 [board]

move3 board = do
  board1 <- move board
  board2 <- move board1
  board3 <- move board2
  return board3
Proof by induction

board1 = move board
board2 = concat (map move board1)

do board1 <- move board
   board2 <- move board1

move board  >>= \board1 ->
mov e board1 >>= \board2 ->

move board  >>= \board1 -> move board1 =
   concat (map (\board1 -> move board1) (move board))
   concat (map move board1)
List comprehension

```
triples = [(x,y,z) |
            z <- [1..], x <- [1..z], y <- [x..z] ]
```

```
triples2 = do z <- [1..]
x <- [1..z]
y <- [x..z]
return (x,y,z)
```

- either one produces
  \[(1,1,1), (1,1,2), (1,2,2), (2,2,2), (1,1,3), \ldots\]

- map is involved, i.e., \(z\) in \(z <- [1..]\) ranges over each list element.
Backtracking

List comprehension can be backtracking.

Lambda can use pattern; calls fail if pattern fails.
MonadPlus

class Monad m => MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a
  msum :: MonadPlus m => [m a] -> m a
  msum = foldr mplus mzero

right zero
  m >>= (\x -> mzero) = mzero
  m `mplus` mzero = m

left zero
  mzero >>= f = mzero
  mzero `mplus` m = m
```
instance MonadPlus Maybe where
  mzero                   = Nothing
  Nothing `mplus` Nothing = Nothing
  Nothing `mplus` Just x  = Just x
  Just x  `mplus` _       = Just x
```

- Check the axioms!
- `msum [Maybe]` finds the first `Just x` or returns `Nothing`. 
List

instance MonadPlus [] where
  mzero = []
  mplus = (++)

- Check the axioms.
- \texttt{msum \texttt{[[a]]}} is the same as \texttt{concat \texttt{[[a]]}}.
### guard

#### For a list:

- `guard :: MonadPlus m => Bool -> m ()`
- `guard True  = return ()`
- `guard False = mzero`

- `guard :: Bool -> [()]`
- `guard True  = [()]`
- `guard False = []`

#### Example:

```hs
pythagoreans3 = do
  z <- [1..]
  x <- [1..z]
  y <- [x..z]
  guard (x^2 + y^2 == z^2)
  return (x, y, z)
```

#### Why?

- `guard` is used to conditionally return values in a monad, effectively creating a list of results based on a boolean condition.