Using Monads to Structure Computation

Jan-Willem Maessen
Laboratory for Computer Science
M.I.T.

Lecture 16

http://www.csg.lcs.mit.edu/6.827

Monadic I/O

**IO a**: computation which does some I/O, then produces a value of type `a`.

`(>>)` :: IO a -> IO b -> IO b

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

`return` :: a -> IO a

Primitive actions:

- `getChar` :: IO Char
- `putChar` :: Char -> IO ()
- `openFile`, `hClose`, ...

Monadic I/O is a clever, type-safe idea which has become very popular in the FL community.

http://www.csg.lcs.mit.edu/6.827
Monadic sequencing

\[ a >>= b \equiv a >>= (\_ \rightarrow b) \]

\[ \text{return } a >>= \lambda x \rightarrow m \equiv (\lambda x \rightarrow m) \ a \]

\[ m >>= \lambda x \rightarrow \text{return } x \equiv m \]

\[ (m >>= \lambda x \rightarrow n) >>= \lambda y \rightarrow o \equiv m >>= \lambda x \rightarrow (n >>= \lambda y \rightarrow o) \quad x \not\in \text{FV}(o) \]

A derived axiom:

\[ m >>= (n >>= o) \equiv (m >>= n) >>= o \]

Syntactic sugar: do

\[ \text{do } e \rightarrow e \]

\[ \text{do } e \ ; \ \text{dostmts} \rightarrow e >>= \lambda \rightarrow \text{do } \text{dostmts} \]

\[ \text{do } \text{return } a \ ; \ \text{dostmts} \rightarrow \text{return } a \ ; \ \text{dostmts} \]

\[ \text{do } \text{return } a \ ; \ \text{dostmts} \rightarrow \text{return } a \ ; \ \text{dostmts} \]

\[ \text{do } \text{let } p=e \ ; \ \text{dostmts} \rightarrow \text{let } p=e \text{ in do } \text{dostmts} \]

\[ \text{do } a \ ; \ b \equiv \text{do } \_ <- a \ ; \ b \]

\[ \text{do } x <- \text{return } a \ ; \ b \equiv (\lambda x \rightarrow \text{do } b) \ a \]

\[ \text{do } x <- \text{return } a \ ; \ b \equiv \text{return } a \ ; \ b \]

\[ \text{do } x <- m \ ; \ \text{return } x \equiv m \]

\[ \text{do } y <- (\text{do } x <- m \ ; \ n) \ ; \ o \equiv \text{do } x <- m \ ; \ (\text{do } y <- n \ ; \ o) \]
Monads and Let

Monadic binding behaves like let:

\[
\begin{align*}
\text{let } x = a \text{ in } m & \equiv (\lambda x \rightarrow m) a \\
\text{let } x = m \text{ in } x & \equiv m \\
\text{let } y = (\text{let } x = m \text{ in } n) \text{ in } o & \equiv \text{let } x = m \text{ in } (\text{let } y = n \text{ in } o) \\
x \not\in \text{FV}(o)
\end{align*}
\]

Monads and Let

- Relationship between monads and let is deep
- Use this to embed languages inside Haskell
- IO is a special sublanguage with side effects

\textbf{class Monad m where}

\textbf{return} :: a \rightarrow m a

\textbf{(>>=)} :: m a \rightarrow (a \rightarrow m b) \rightarrow m b

\textbf{(>>)} :: m a \rightarrow m b \rightarrow m b

\textbf{fail} :: String \rightarrow m a

\textbf{--*}
Outline

- Monadic operations and their properties
- Reasoning about monadic programs
- Creating our own monads:
  - Id: The simplest monad
  - State
  - Supplying unique names
  - Emulating simple I/O
  - Exceptions
- Composing monad transformers
- IO and ST: two very special monads
- Using ST for imperative computation
- Ordering issues

Proving simple properties

```haskell
putString []    = return ()
putString (c:cs) = putChar c >> putString cs

[]     ++ bs    = bs
(a:as) ++ bs    = a : (as ++ bs)
```

Show:

```haskell
putString as >> putString bs
≡ putString (as++bs)
```
Base case

putString [] = return ()

[] ++ bs = bs

putString [] >> putString bs
≡ return () >> putString bs
≡ putString bs
≡ putString ([]++bs)

Inductive case

putString (a:as) = putChar a >> putString as

(a:as) ++ bs = a : (as ++ bs)

putString (a:as) >> putString bs
≡ (putChar a >> putString as) >> putString bs
≡ putChar a >> (putString as >> putString bs)
≡ putChar a >> (putString (as ++ bs))
≡ putString (a : (as ++ bs))
≡ putString ((a:as) ++ bs)
Representation Independence

- Our proof did not depend on the behavior of I/O!
- Uses properties of Monads
- Requires some function
  
  \[
  \text{putChar :: Char} \rightarrow \text{m ()}
  \]

A monadic computation has two sets of operations:
- The monadic operations, with general properties
- Specific operations with unique properties

Fib in Monadic Style

\[
\text{fib n = if } (n \leq 1) \text{ then } n \text{ else let }
\begin{align*}
  n1 &= n - 1 \\
  n2 &= n - 2 \\
  f1 &= \text{fib } n1 \\
  f2 &= \text{fib } n2 \\
  \text{in } f1 + f2
\end{align*}
\]

\[
\text{fib n = if } (n \leq 1) \text{ then } n \text{ else do}
\begin{align*}
  n1 &= \text{return } (n-1) \\
  n2 &= \text{return } (n-2) \\
  f1 &= \text{fib } n1 \\
  f2 &= \text{fib } n2 \\
  \text{return } (f1+f2)
\end{align*}
\]

Note the awkward style: everything must be named!
The Simplest Monad

newtype Id a = Id a

instance Monad Id where
  return a = Id a
  Id a >>= f = f a

runId (Id a) = a

• This monad has no special operations!
• Indeed, we could just have used \texttt{let}
• The \texttt{runId} operation runs our computation

The State Monad

• Allow the use of a single piece of mutable state

  \texttt{put} :: s \to State s ()
  \texttt{get} :: State s s

  \texttt{runState} :: s \to State s r \to (s, r)

  instance Monad (State s)
Generating Unique Identifiers

type Uniq = Int
type UniqM = State Int

runUniqM :: UniqM r -> r
runUniqM comp = snd (runState 0 comp)

uniq :: UniqM Uniq
uniq = do u <- get
         put (u+1)
         return u

State

newtype State s r = S (s -> (s,r))

instance Monad (State s) where
  return r = S (\s -> (s,r))
  S f >>>= g = S (\s -> let (s', r) = f s
                     S h = g r
                     in  h s')

get      = S (\s -> (s,s))
put s    = S (\o -> (s, ())
runState s (S c) = c s
Poor Man’s I/O

type PoorIO a = State (String, String)

putChar :: Char -> PoorIO ()
putChar c = do (in, out) <- get
  put (in, out++[c])

getChar :: PoorIO Char
getChar = do (in, out) <- get
  case in of
    a:as -> do put (as, out)
    _    -> fail "EOF"

Error Handling using Maybe

instance Monad Maybe where
  return a = Just a
  Nothing >>= f = Nothing
  Just a >>= f = f a
  fail _   = Nothing

  Just a `mplus` b = Just a
  Nothing `mplus` b = b

  do m' <- matrixInverse m
     y <- matrixVectMult m x
     return y
Combining Monads

- To simulate I/O, combine State and Maybe.
- There are two ways to do this combination:

```haskell
newtype SM s a = SM (s -> (s, Maybe a))
newtype MS s a = MS (s -> Maybe (s, a))

SM MS
([],"") ([])"

[do putChar 'H'
  a <- getChar ([])"
  putChar 'I'
  `mplus` putChar '!']
  ([])"
  ([])"
```

Monad Transformers

- State and error handling are separate features
- We can plug them together in multiple ways
- Other monads have a similar flavor
- Monad Transformer: add a feature to a Monad.

```haskell
instance (Monad m) => Monad (ErrorT m)
instance (Monad m) => Monad (StateT s m)

type ErrorM = ErrorT Id

type StateM s = StateT s Id

type SM s a = StateT s (ErrorT Id)

type MS s a = ErrorT (StateT s Id)
```
Special Monads

- Operations inexpressible in pure Haskell
- IO Monad
  Primitives must actually call the OS
  Also used to embed C code
- State Transformer Monad
  Embeds arbitrary mutable state
  Alternative to M-structures + barriers

The State Transformer Monad

```haskell
instance Monad (ST s)

newSTRef :: a -> ST s (STRef s a)
readSTRef :: STRef s a -> ST s a
writeSTRef :: STRef s a -> a -> ST s ()

runST :: (∀s. ST s a) -> a
```

- The special type of `runST` guarantees that an `STRef` will not escape from its computation.
Independent State Transformers

- In **ST s t**, the type **s** represents the “world.”
- We can have multiple independent worlds.
- The type of **runST** keeps them from interacting.

---

Mutable lists using ST

We can create as many mutable references as we like, allowing us to build mutable structures just as we would with I- and M-cells.

```haskell
data RList s t = RNil
              | RCons t (STRef s t)

rCons :: t -> RList s t -> ST s (RList s t)
rCons t ts = do r <- newSTRef ts
                return (RCons t r)
```

---
Insert using RList

\[
\text{insertr RNil} \quad x = \text{rCons} \ x \ \text{RNil} \\
\text{insertr} \ ys@(\text{RCons} \ y \ yr) \ x = \\
\quad \text{if} \ x==y \ \text{then return} \ ys \\
\quad \text{else do} \ ys' \ <- \ \text{readSTRef} \ yr \\
\quad \quad ys'' \ <- \ \text{insertr} \ ys' \ x \\
\quad \quad \text{writeSTRef} \ yr \ ys'' \\
\quad \text{return} \ ys
\]

Graph traversal: \textit{ST notebook}

\[
\text{data} \ \text{GNode} = \text{GNode} \ \text{NodeId} \ \text{Int} \ \text{[GNode]} \\
\text{rsum} \ \text{node} = \text{do} \\
\quad \text{nb} \ <- \ \text{mkNotebook} \\
\quad \text{let} \ \text{rsum'} \ (\text{GNode} \ x \ i \ \text{nbs}) = \text{do} \\
\quad \quad \text{seen} \ <- \ \text{memberAndInsert} \ \text{nb} \ x \\
\quad \quad \text{if} \ \text{seen} \\
\quad \quad \quad \text{then return} \ 0 \\
\quad \quad \text{else do} \ \text{nbs'} \ <- \ \text{mapM} \ \text{rsum'} \ \text{nbs} \\
\quad \quad \quad \text{return} \ (i + \text{sum} \ \text{nbs'})
\]
A traversal notebook

type Notebook s = STRef s (RList s Nodeid)

mkNotebook = newSTRef RNil

memberAndInsert nb id = do
  ids <- readSTRef nb
  case ids of
    MNil -> do t <- rCons id MNil
              writeSTRef nb t
              return False
    MCons id' nb'
      | id==id' = return True
      | otherwise = memberAndInsert nb' id


Problems with Monadic Style

- We need a new versions of common functions:

  mapM f [] = return []
  mapM f (x:xs) = do
    a <- f x
    as <- mapM f xs
    return (a:as)

  mapM' f [] = return []
  mapM' f (x:xs) = do
    as <- mapM' f xs
    a <- f x
    return (a:as)
Monads and Ordering

• Monads aren’t inherently ordered (Id)
• But stateful computations must be ordered
• For ST and IO, at least the side-effecting computations are ordered.
• The unsafeInterleaveIO construct relaxes this ordering, but is impure.

• On the other hand, barriers order all computation, including non-mondic execution.

There is still room for experimentation!