Monad class

Motivation:
- Separation of pure and impure code
- Properties of a particular kind of functions
- Introduction of state and its transformations
- Or simply: yet another type class

For more material: see the course web.

The functional IO problem

IO has traditionally been included as side effects to ordinary functions:

inputInt :: Int
then substituting equals for equals is no longer possible

inputDiff = inputInt - inputInt

probably not equal to zero

Example: Lambda laughter

Suppose the function:

outputChar :: char -> ()

has the side effect of printing its argument. The expression (note: pseudo-Haskell):

outputChar 'h'; outputChar 'a';
outputChar 'h'; outputChar 'a'

Should then print the string "haha".
Lambda laughter, cont.

If we then try to catch the repetitive pattern by instead writing:

```
let x = (outputChar 'h'; outputChar 'a') in
  x; x
```

"then the laugh is on us" (P. Wadler). It will print only "ha". Why?

However, the following (note: pseudo-Haskell):

```
let f() = (outputChar 'h'; outputChar 'a') in
  f(); f()
```

will print the string "haha".

The IO Monad

The type of IO activities:

```
prompt :: IO ()
prompt = putStr ">:"
```

The type of IO activities that return a value:

```
getChar :: IO Char
getLine :: IO String
```

The do-notation

Allows sequencing and naming the returned values:

```
echoReverse :: IO ()
echoReverse = do
  aLine <- getLine
  putStrLn (reverse aLine)
```
### return and let

The `return` operation does the empty IO-activity:

```haskell
ggetInt :: IO Int
gGetInt = do
  aLine <- getLine
  return (read aLine :: Int)
```

Local variables may be defined using `let`:

```haskell
echoReverse2 :: IO ()
echoReverse2 = do
  aLine <- getLine
  let theLineReversed = reverse aLine
  putStrLn (theLineReversed)
```

### Imperative style

One can e.g. use the conditional clause:

```haskell
testPalindrome :: IO ()
testPalindrome = do
  prompt
  aLine <- getLine
  if (aLine == (reverse aLine)) then
    putStrLn "Yes, a palindrome."
  else
    putStrLn "No, no!"
```

### Loops done with recursion

```haskell
testPalindroms :: IO ()
testPalindroms = do
  prompt
  aLine <- getLine
  if (aLine == "") then
    return ()
  else do
    if (aLine == (reverse aLine)) then
      putStrLn "Yes, a palindrome."
    else
      putStrLn "No, no!"
    testPalindroms
```

### Monadic lambda laughter

The IO monad gives type-safe laughers:

```haskell
laugh :: IO ()
laugh =
  let x = do putChar 'h'; putChar 'a'
  in do x; x
```
IO-stripping

IO stripping is not allowed (at least officially:-) because if you could strip off side effects with:

\[
\text{stripIO :: IO \ a \to \ a}
\]

then the following code

\[
\text{inputInt :: Int} \\
\text{inputInt = ioStrip getInt}
\]

\[
\text{inputDiff = inputInt - inputInt}
\]

would violate “equals for equals” principle.

Back door

In the module System.IO.Unsafe there actually is

\[
\text{unsafePerformIO :: IO \ a \to \ a}
\]

behaving as expected. However:

- it is NOT type-safe;
- you could coerce any type to any other type using unsafePerformIO!

FORGET IT IMMEDIATELY!

Monad class (again)

Motivation:

- Separation of pure and impure code
- Properties of a particular kind of functions
- Introduction of state and its transformations
- Or simply: yet another type class

Composing functions

Composing functions is simple:

\[
f :: a \to b \\
g :: b \to c
\]

then \( g \) and \( f \) may be composed to \( g \cdot f \).

But suppose:

\[
f :: a \to \text{Maybe } b \\
g :: b \to \text{Maybe } c
\]

then how to compose \( g \) and \( f \)? And what will it mean?
Composing functions

Yet another example:

children :: Person -> [Person]
grandchildren :: Person -> [Person]

then almost

grandchildren = children.children

but not exactly.

More functors

An example of its utility:

import Data.Char
import Data.List
fmaptest = do
  line <- fmap (intersperse '-' . reverse . map toUpper)
  putStrLn line

putStrLn line

More functors

An example of its utility:

import Data.Char
import Data.List
fmaptest = do
  line <- fmap (intersperse '-' . reverse . map toUpper)
  putStrLn line

putStrLn line

Slightly more complicated:

ghci> let a = fmap (*) [1,2,3,4]
ghci> :t a
a :: [Integer -> Integer]
ghci> fmap (\f -> f 9) a
[9,18,27,36]
Applicative functors

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

instance Applicative Maybe where
  pure = Just
  Nothing <*> _ = Nothing
  (Just f) <*> something = fmap f something

Some more ...

So we can apply:

pure fn <*> x <*> y <*> ...

Considering that pure fn <*> x equals fmap fn x we can write instead:

fmap fn x <*> y <*> ...

or even more cleanly:

(<$>) :: (Functor f) => (a -> b) -> f a -> f b
fn <$> x = fmap fn x

fn <$> x <*> y <*> ...

ghci> [(+),(*)] <*> [1,2] <*> [3,4,5,6,3,4,6,8]

Rounding up

instance Applicative [] where
  pure x = [x]
  fs <*> xs = [f x | f <- fs, x <- xs]

instance Applicative IO where
  pure = return
  a <*> b = do
    f <- a
    x <- b
    return (f x)
The Monad class

class Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  (>>) :: m a -> m b -> m b
  return :: a -> m a
  fail :: String -> m a

-- Minimal complete definition:
-- (>>=), return
  m >>= k = m >>= \_ -> k
  fail s = error s

Operations >>= (bind) and >> are (by some) pronounced “then”.

Just a word on return, again

return is a bad name!

main = do
  s <- getLine
  return ()
  putStrLn s

works as charm!

The MonadPlus class

class (Monad m) => MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a

Algebraically: a monoid

The identity monad

data Id a = Id a

instance Monad Id where
  return x = Id x
  (Id x) >>= f = f x
The List monad

instance Monad [] where
  return x = [x]
  xs >>= f = concat (map f xs)
  fail s = []

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

The Maybe monad

instance Monad Maybe where
  return x = Just x
  Just x >>= f = f x
  Nothing >>= f = Nothing

instance MonadPlus Maybe where
  mzero = Nothing
  Nothing `mplus` ys = ys
  xs `mplus` ys = xs

The do-notation

do-expressions are just syntactic sugar for >>= or >=

echoReverse = do
  aLine <- getLine
  putStrLn (reverse aLine)
  is just

echoReverse =
  getLine >>= \aLine ->
  putStrLn (reverse aLine)

List comprehensions

"Reverse" do-notation:

list1 = [(x, y) | x <- [1..], y <- [1..x]]

list2 = do
  x <- [1..]
  y <- [1..x]
  return (x, y)
Monad class

Motivation:
- Separation of pure and impure code
- Properties of a particular kind of functions
- Introduction of state and its transformations
- Or simply: yet another type class

A simple random number generator:

\[ g : \text{Integer} \rightarrow (\text{Float}, \text{Integer}) \]
\[ g\ \text{seed} = (\text{fromInteger(newSeed)}/\text{fromInteger(m)}, \text{newSeed}) \]
where
\[ \text{newSeed} = (\text{seed}\times a) \mod m \]
\[ a = 1812433253 \]
\[ m = 2^{32} \]
\[ \text{initialSeed} = 3121281023 \]

Usage

(a, s1) = g initialSeed
(b, s2) = g s1
(c, s3) = g s2
(xs, s4) = (values, last seeds)
where (values, seeds) =
  (unzip.take 17.tail) (iterate (g.snd) (dummy, s3))
  dummy = 45.0

Quite clumsy.

A generalisation attempt

newtype RandomGenerator a = Ran (Integer -> (a, Integer))
random = Ran g
generate (Ran f) = (fst.f) initialSeed
The Random monad

type R = RandomGenerator

instance Monad RandomGenerator where
  return :: a -> R a
  return x = Ran (seed -> (x, seed))

  (>>=) :: R a -> (a -> R b) -> R b
  (Ran g0) >>= f = Ran (seed -> let (y, seed1) = g0 seed
                         in g1 seed1)

randoms3 = do
  a <- random
  b <- random
  c <- random
  return (a, b, c)

result3 = generate randoms3

randomList 0 = return []
randomList n = do
  x <- random
  xs <- randomList (n-1)
  return (x:xs)

randomPair = do
  a <- random
  b <- random
  return (a, b)

is equivalent to

randomPair =
  random >>= \a -> random >>= \b -> return (a, b)

Or

randomPair = random >>= (\a -> random >>= (\b -> return (a, b)))
Another state transformation example

type Dictionary = [(String, String)]
dictAdd key value dict = (key, value):dict
dictFind key dict = lookup key dict

Passing state without side effects is clumsy:

result1 = r where
  d1 = dictAdd "no" "norway" []
  d2 = dictAdd "se" "sweden" d1
  r1 = dictFind "fr" d2
  d3 = dictAdd "fr" "france" d2
  r = dictFind "fr" d3

The StateTransform monad

newtype StateTransform s a = ST (s -> (s, a))
apply (ST f) = f

instance Monad (StateTransform s) where
  return x = ST $ \s -> (s, x)
x >>= f = ST $ \s0 ->
    let (s1, y) = apply x s0
    in (apply (f y)) s1

stateUpdate :: (s -> s) -> StateTransform s ()
stateUpdate u = ST $ \s -> (u s, ())
stateQuery :: (s -> a) -> StateTransform s a
stateQuery q = ST $ \s -> (s, q s)
runST s t = apply t s

State passing becomes invisible: robustness.

Dictionary example again

type DictMonad = StateTransform Dictionary

result2 = snd (runST [] dictM) where
  dictM :: DictMonad (Maybe String)
dictM = do
  stateUpdate (dictAdd "no" "norway")
  stateUpdate (dictAdd "se" "sweden")
r1 <- stateQuery (dictFind "fr")
  stateUpdate (dictAdd "fr" "france")
r <- stateQuery (dictFind "fr")

Monad class

Motivation:
- Separation of pure and impure code
- Properties of a particular kind of functions
- Introduction of state and its transformations
- Or simply: yet another type class
Kleisli triple

- A type construction: for type $a$ create $Ma$
- A unit function $a \rightarrow Ma$ (return in Haskell)
- A binding operation of polymorphic type $Ma \rightarrow (a \rightarrow Mb) \rightarrow Mb$. Four stages (informally):
  - The monad-related structure on the first argument is "pierced" to expose any number of values in the underlying type $a$.
  - The given function is applied to all of those values to obtain values of type $(M b)$.
  - The monad-related structure on those values is also pierced, exposing values of type $b$.
  - Finally, the monad-related structure is reassembled over all of the results, giving a single value of type $(M b)$.

Monad axioms:

- $return$ acts as a neutral element of $\ggg$.
  - $(return \ x) \ggg f \leftrightarrow f \ x$
  - $m \ggg return \leftrightarrow m$

- Binding two functions in succession is the same as binding one function that can be determined from them.
  - $(m \ggg f) \ggg g \leftrightarrow m \ggg \lambda x.(f \ x \ggg g)$

We can restate it in a cleaner way (Thomson). Define

$$ (>\circ >) :: \text{Monad } m \Rightarrow (a \rightarrow m b) \rightarrow (b \rightarrow m c) \rightarrow (a \rightarrow m c)$$

$$f >\circ > g = \lambda x \rightarrow (f \ x) \ggg g$$

Now, the monad axioms may be written as:

$$\text{return} \ >\circ > f = f$$
$$f >\circ > \text{return} = f$$

$$(f >\circ > g) >\circ > h = f >\circ > (g >\circ > h)$$

Example

$$a = \text{do } x \leftarrow [3..4]$$
$$[1..2]$$
$$\text{return } (x, 42)$$

is equivalent to

$$a = [3..4] \ggg (\lambda x \rightarrow [1..2] \ggg (\_ \rightarrow \text{return } (x, 42)))$$

Thus, remembering that

$$\text{instance Monad } [] \text{ where}$$
$$m \ggg f = \text{concatMap } f \ m$$
$$\text{return } x = [x]$$
$$\text{fail } s = []$$
The transformations may be reduced as follows:

```
a = [3..4] >>= (\x -> [1..2] >>= (\_ -> return (x, 42)))
a = [3..4] >>=
      (\x -> concatMap (\_ -> return (x, 42)) [1..2])
a = [3..4] >>= (\x -> [(x,42),(x,42)])
a = concatMap (\x -> [(x,42),(x,42)]) [3..4]
a = [(3,42),(3,42),(4,42),(4,42)]
```