**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:

```haskell
inputInt :: Int
```

then substituting equals for equals is no longer possible

```haskell
inputDiff = inputInt - inputInt
```

probably not equal to zero

---

**Example: Lambda laughter**

Suppose the function:

```haskell
outputChar :: char -> ()
```

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

```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:

```haskell
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):

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

will print the string "haha".

The IO Monad

The type of IO activities:

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

The type of IO activities that return a value:

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

The do-notation

Allows sequencing and naming the returned values:

```haskell
echoReverse :: IO ()
echoReverse = do
    aLine <- getLine
    putStrLn (reverse aLine)
```
The \texttt{return} operation does the empty IO-activity:

\begin{verbatim}
gGetInt :: IO Int
gGetInt = do
    aLine <- getLine
    return (read aLine :: Int)
\end{verbatim}

Local variables may be defined using \texttt{let}:

\begin{verbatim}

echoReverse2 :: IO ()
echoReverse2 = do
    aLine <- getLine
    let theLineReversed = reverse aLine
    putStrLn (theLineReversed)
\end{verbatim}

One can e.g. use the conditional clause:

\begin{verbatim}
testPalindrome :: IO ()
testPalindrome = do
    prompt
    aLine <- getLine
    if (aLine == (reverse aLine)) then
        putStrLn "Yes, a palindrome."
    else
        putStrLn "No, no!"
\end{verbatim}

The IO monad gives type-safe laughters:

\begin{verbatim}

laugh :: IO ()
laugh =
    let x = do putChar 'h'; putChar 'a'
    in do x; x
\end{verbatim}
**IO-stripping**

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

```
stripIO :: IO a -> a
```

then the following code

```
inputInt :: Int
inputInt = ioStrip getInt
```

```
inputDiff = inputInt - inputInt
```

would violate “equals for equals” principle.

**Back door**

In the module `System.IO.Unsafe` there actually is

```
unsafePerformIO :: IO a -> 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 -> b
```

```
g :: b -> c
```

then `g` and `f` may be composed to `g . f`.

But suppose:

```
f :: a -> Maybe b
```

```
g :: b -> Maybe c
```

then how to compose `g` and `f`? And what will it mean?
Composing functions

Yet another example:

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

then almost

```haskell
grandchildren = children.children
```

but not exactly.

Functor (reminder)

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

Please note that

```haskell
instance Functor IO where
    fmap f action = do
        result <- action
        return (f result)
```

More functors

An example of its utility:

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

Slightly more complicated:

```haskell
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

Applicative functors: examples

ghci> Just (+3) <*> Just 9
Just 12
ghci> pure (+3) <*> Just 10
Just 13
ghci> pure (+3) <*> Just 9
Just 12
ghci> Just ("hahah") <*> Nothing
Nothing
ghci> Nothing <*> Just "woot"
Nothing
ghci> pure (+) <*> Just 3 <*> Just 5
Just 8
ghci> pure (+) <*> Just 3 <*> Nothing
Nothing
ghci> pure (+) <*> Nothing <*> Just 5
Nothing

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 <*> y <*> ...

ghci> [(+),(*)] <$> [1,2] <*> [3,4]
[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

```haskell
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!

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

works as charm!

The MonadPlus class

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

Algebraically: a monoid
```

The identity monad

```haskell
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

From the assignment (by atp08mla):

```
singleWildcardMatch (wc:ps) (x:xs) = match wc ps xs >> Just [x]
```

**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)
```
**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:**

\[
g : \text{Integer} \rightarrow (\text{Float, Integer})
g \text{seed} = (\text{fromInteger(newSeed)}/\text{fromInteger(m)}, \text{newSeed})
\]

where

\[
\text{newSeed} = (\text{seed} \times a) \mod m
\]

\[
a = 1812433253
\]

\[
m = 2^{32}
\]

\[
\text{initialSeed} = 3121281023
\]

**Usage**

\[
\begin{align*}
(a, s1) &= g \text{ initialSeed} \\
(b, s2) &= g \text{ s1} \\
(c, s3) &= g \text{ s2} \\
(xs, s4) &= (\text{values, last seeds})
\end{align*}
\]

where (values, seeds) =

\[
(\text{unzip}.\text{take 17}.\text{tail}) (\text{iterate} (g.\text{snd}) (\text{dummy, s3}))
\]

\[
\text{dummy} = 45.0
\]

Quite clumsy.

**A generalisation attempt**

```haskell
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
                          (Ran g1) = f y
                          in g1 seed1)

Using the random monad

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)

do-notation, again

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))

The pattern

State transformation and value generation
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

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")

State passing becomes invisible: robustness.

The StateTransform monad

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

instance Monad (StateTransform s) where
  return x = ST $ \
  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 -> (q s)
runST s t = apply t s

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 → Ma` (`return` in Haskell)
- **A binding operation** of polymorphic type `Ma → (a → Mb) → Mb`. Four stages (informally):
  1. The monad-related structure on the first argument is "pierced" to expose any number of values in the underlying type `a`.
  2. The given function is applied to all of those values to obtain values of type `(M b)`.
  3. The monad-related structure on those values is also pierced, exposing values of type `b`.
  4. 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 `>>>`.
  
  \[(\text{return } x) \ggg f ⇔ f x\]

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

**Example**

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

\[(\ggg) :: \text{Monad } m \Rightarrow (a -> m b) -> (b -> m c) -> (a -> m c)\]

\[f \ggg g = \backslash x -> (f x) \ggg g\]

Now, the monad axioms may be written as:

- `return >@> f = f`
- `f >@> return = f`
- `(f >@> g) >@> h = f >@> (g >@> h)`

```
instance Monad [] where
  m >>> f = concatMap f m
  return x = [x]
  fail s = []
```

```haskell
a = do x <- [3..4]
    [1..2]
    return (x, 42)
```

is equivalent to

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

Thus, remembering that
The transformations may be reduced as follows:

\[
a = [3..4] >>= (\ x \to \ [1..2] >>= (\ _ \to \ \text{return} (x, 42)))
\]

\[
a = [3..4] >>= (\ x \to \ \text{concatMap} (\ _ \to \ \text{return} (x, 42)) [1..2])
\]

\[
a = [3..4] >>= (\ x \to \ [(x,42),(x,42)])
\]

\[
a = \text{concatMap} (\ x \to \ [(x,42),(x,42)]) [3..4]
\]

\[
a = [(3,42),(3,42),(4,42),(4,42)]
\]