Lecture 13. More monads and applicatives

Functional Programming 2018/19

Alejandro Serrano

Goals

- See yet another example of *monad*
- Introduce the idea of *applicative* functor

Chapter 12.2 from Hutton's book

The State monad

Reverse Polish Notation (RPN)

Notation in which an operator follows its operands

```
   3 4 + 2 * 10 -
=      7 2 * 10 -
=         14 10 -
=               4
```

Parentheses are not needed when using RPN

Historical note: RPN was invented in the 1920s by the Polish mathematician Łukasiewicz, and rediscovered by several computer scientists in the 1960s

RPN expressions

Expressions in RPN are lists of numbers and operations

```
data Instr = Number Float | Operation ArithOp
type RPN   = [Instr]
```

We reuse the `ArithOp` type from arithmetic expressions

For example, `3 4 + 2 *` becomes

```
[ Number 3, Number 4, Operation Plus
, Number 2, Operation Times ]
```

RPN calculator

To compute the value of an expression in RPN, you keep a stack of values

- Each number is added at the top of the stack
- Operations use the top-most elements in the stack

```
             3         4         +         2         *

                     | 4 |               | 2 |
          -> | 3 | -> | 3 | -> | 7 | -> | 7 | -> | 14 |
+--+        +---+     +---+     +---+     +---+     +----+
```

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Case study: RPN calculator

```
type Stack = [Float]

evalInstr :: Instr -> Stack -> Stack
evalInstr (Number f)         stack
      = f : stack
evalInstr (Operation op) (x:y:stack)
      = evalOp op x y : stack
        where evalOp ...
```

Case study: RPN calculator

Let me introduce two new operations to make clear what is going in with the stack

```
pop :: Stack -> (Float, Stack)
pop (x:xs) = (x, xs)
push :: Float -> Stack -> Stack
push x xs  = x : xs
```

Using those the evaluator takes this form:

```
evalInstr (Number f)     s
  = push f s
evalInstr (Operation op) s
  = let (x, s1) = pop s
        (y, s2) = pop s1
    in push (evalOp op x y) s2
```

Encoding state explicitly

A function like pop

```
pop :: Stack -> (Float, Stack)
```

can be seen as a function which modifies a state:

- Takes the original state as an argument
- Returns the new state along with the result

The intuition is the same as looking at IO as

```
type IO a = World -> (a, World)
```

Encoding state explicitly

Functions which only operate in the state return ()

```
push       :: Float -> Stack -> ((), Stack)
push f s = ((), f : s)

evalInstr :: Instr -> Stack -> ((), Stack)
evalInstr (Number f)     s
  = push f s
evalInstr (Operation op) s
  = let (x, s1) = pop s
        (y, s2) = pop s1
    in push (evalOp op x y) s2
```

Looking for similarities

The same pattern occurs twice in the previous code

```
let (x, newStack) = f oldStack
in  _ -- something which uses x and the newStack
```

This leads to a higher-order function

```
next :: (Stack -> (a, Stack))
     -> (a -> Stack -> (b, Stack))
     -> (Stack -> (b, Stack))
next f g = \s -> let (x, s') = f s
                 in  g x s'
```

(Almost) the State monad

```
type State a = Stack -> (a, Stack)
```

`State` is almost a monad, we only need a `return`

- The type has only one hole, as required

The missing part is a `return` function

- The only thing we can do is keep the state unmodified

```
return :: a -> Stack -> (a, Stack)
return x = \s -> (x, s)
```

Nicer code for the examples

```haskell
evalInstr :: Inst -> State ()
...
evalInstr (Operation op)
  = do x <- pop
       y <- pop
       push (evalOp x y)
...
```

The `Stack` value is threaded implicitly

- Similar to a single mutable variable

Notes on implementation

We can generalize this idea to any type of `State`

```
type State s a = s -> (a, s)
```

Alas, if you try to write the instance GHC complains

```
instance Monad (State s) where  -- Wrong!
```

This is because you are only allowed to use a type synonym with *all* arguments applied

- But you need to leave one out to make it a monad

Notes on implementation

The “trick” is to wrap the value in a data type

```
data State s a = S (s -> (a, s))

run :: State s a -> s -> a
run (S f) s = fst (f s)
```

But now every time you need to access the function, you need to unwrap things, and then wrap them again

```
instance Monad (State s) where
  return x = S $ \s -> (x, s)
  (S st) >>= g = S $ \s -> let (x, s') = f s
                               S g' = g x
                           in g s'
```

What is going on?

Warning: the following slides contain ASCII-art

What is going on?

A `State s a` value is a "box" which, once feed with an state, gives back a value and the modified state

```
        +--+ --> v
        |  |
s --> +--+ --> s'
```

What is going on?

A `State s a` value is a “box” which, once feed with an state, gives back a value and the modified state

```
        +--+ --> v
        |  |
s -->   +--+ --> s'
```

A function `c -> State s a` is a “box” with an extra input

```
c -->   +--+ --> v
        |  |
s -->   +--+ --> s'
```

What is going on with `return`?

`return` has type `a -> State s a`

- It is thus a box of the second kind
- It just passes the information through, unmodified

```
x --> +--------+ --> x
      | return |
s --> +--------+ --> s
```

What is going on with (>>=)?

```
(>>=) : State s a -> (a -> State s b) -> State s b
```

- We take one box of each kind
- And have to produce a box of the second kind

```
     +-----+ --> a         a --> +---+ --> b
     | st  |                     | g |
s --> +-----+ --> s'       s --> +---+ --> s'
```

Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

What is going on with `(>>=)`?

```
(>>=) : State s a -> (a -> State s b) -> State s b
```

- We take one box of each kind
- And have to produce a box of the second kind

```
      +----+ --> a        a --> +---+ --> b
      | st |                    | g |
s --> +----+ --> s'       s --> +---+ --> s'
```

Connect the wires and wrap into a larger box!

```
   +-----------------------------------+
   |  +----+ ----------------> +---+ --> b
   |  | st |                   | g |  |
s --> +----+ ----------------> +---+ --> s'
   +-----------------------------------+
```

Another use for state: counters

Given a binary tree, return a new one labelled with numbers in depth-first order

```
> let t = Node (Node Leaf 'a' Leaf)
               'b'
               (Node Leaf 'c' Leaf)
> label t
Node (Node Leaf (0, 'a') Leaf)
     (1, 'b')
     (Node Leaf (2, 'c') Leaf)
```

The type for such a function is

```
label :: Tree a -> Tree (Int, a)
```

Idea: use an implicit counter to keep track of the label

Cooking `label`

The main work happens in a local function which is stateful

```
label' :: Tree a -> State Int (Tree (Int, a))
```

The purpose of `label` is to initialize the state to `0`

```
label t = run (label' t) 0
  where label' = ...
```

Cooking `label'`

We use an auxiliary function to get the current label and update it to the next value

```
nextLabel :: State Int Int
nextLabel = S $ \i -> (i, i + 1)
```

Armed with it, writing the stateful `label'` is easy

```
label' Leaf         = return Leaf
label' (Node l x r) = do l' <- label' l
                         i  <- nextLabel
                         r' <- label' r
                         return (Node l' (i, x) r')
```

Monad laws

As with functors, valid monads should obbey some laws

```
-- return is a left identity
do y <- return x   ==   f x
   f y
-- return is a right identity
do x <- m          ==   m
   return x
-- bind is associative
do y <- do x <- m     do x <- m          do x <- m
           f x    ==     do y <- f x ==     y <- f x
   g y                   g y                g y
```

In fact, monads are a higher-order version of monoids

Summary of monads

Different monads provide different capabilities

- `Maybe` monad models optional values and failure
- `State` monad threads an implicit value
- `[]` monad models search and non-determinism
- `IO` monad provides impure input/output

Summary of monads

Different monads provide different capabilities

- `Maybe` monad models optional values and failure
- `State` monad threads an implicit value
- `[]` monad models search and non-determinism
- `IO` monad provides impure input/output

There are even more monads!

- `Either` models failure, but remembers the problem
- `Reader` provides a read-only environment
- `Writer` computes an on-going value
  - For example, a log of the execution
- `STM` provides atomic transactions

Summary of monads

Monads provide a common interface

- `do`-notation is applicable to all of them
- Many utility functions (to be described)

Applicatives

Lifting functions

When explaining `Maybe` and `IO` we introduced `liftM2`

```
liftM2 :: (a -> b -> c)
       -> Maybe a -> Maybe b -> Maybe c
liftM2 :: (a -> b -> c)
       -> IO    a -> IO    b -> IO    c
```

In general, we can write `liftM2` for any monad

```
liftM2 :: Monad m => (a   -> b   -> c)
                  ->  m a -> m b -> m c
liftM2 f x y = do x' <- x
                  y' <- y
                  return (f x' y')
```

Lifting functions

This makes the code shorter and easier to read

```
-- Using do notation
do fn' <- validateFirstName fn
   ln' <- validateLastName  fn
   return (Person fn' ln')

-- Using lift
liftM2 Person (validateFirstName fn)
              (validateLastName  ln)
```

Lifting with different number of arguments

```
liftM1 :: (a -> b) -> m a -> m b
liftM3 :: (a -> b -> c -> d)
       -> m a -> m b -> m c -> m d
liftM4 :: ...
```

Lifting with different number of arguments

```
liftM1 :: (a -> b) -> m a -> m b
liftM3 :: (a -> b -> c -> d)
       -> m a -> m b -> m c -> m d
liftM4 :: ...
```

The implementation of `liftM` follows the same pattern

```
liftM3 f x y z = do x' <- x
                    y' <- y
                    z' <- z
                    return (f x' y' z')
```

Could you find a nicer implementation for `liftM1`?

Lifting with different number of arguments

```
liftM1 :: (a -> b) -> m a -> m b
liftM3 :: (a -> b -> c -> d)
       -> m a -> m b -> m c -> m d
liftM4 :: ...
```

The implementation of `liftM` follows the same pattern

```
liftM3 f x y z = do x' <- x
                    y' <- y
                    z' <- z
                    return (f x' y' z')
```

Could you find a nicer implementation for `liftM1`?

```
liftM1 = fmap
```

Lifting with different number of arguments

This is clearly suboptimal:

- We need to provide different `liftM` with almost the same implementation
- If we refactor the code by adding or removing parameters to a function, we have to change the `liftM` function we use at the call site

Can we do better?

Introducing (<*>)

Suppose we want to lift a function with two arguments:

```
f :: a -> b -> c    x :: f a    y :: f b
```

What happens if we `fmap` it?

```
fmap f :: f a -> f (b -> c)
```

We are able to apply the first argument

```
fmap f x :: f (b -> c)
```

The result is not in the form we want

- The function is now *inside* the functor/monad

Introducing (<*>)

To apply the next argument we need some magical function

```
(<*>) :: f (b -> c) -> f b -> f c
```

If we had that function, then we can write

```
fmap f x <*> y
= -- using the synonym (<$>) = fmap
f <$> x <*> y
```

Introducing (<*>)

```
(<*>) :: f (b -> c) -> f b -> f c
```

Note that in the type of `(<*>)` we can choose `c` to be yet another function type

- As a result, by means of `fmap` and `(<*>)` we can lift a function with any number of arguments

```
f  :: a -> b -> ... -> y -> z
ma :: m a
mb :: m b
...
f <$> ma <*> mb <*> ... <*> my :: m z
```

Using (<*>)

Take the `label'` functions for trees we wrote previously

```
label' Leaf           = return Leaf
label' (Node l x r) = do l' <- label' l
                         i  <- nextLabel
                         r' <- label' r
                         return (Node l' (i, x) r')
```

Now we would write instead:

```
label' Leaf = return Leaf
label' (Node l x r)
        = Node <$> label' l
               <*> ( (,x) <$> nextLabel )
               <*> label' r
```

Applicatives

It turns out that `(<*>)` by itself is an useful abstraction

- `Functor` allows you to lift one-argument function
- With `(<*>)` you can lift functions with more than one argument

For completeness, we also want a way to lift 0-ary functions

A type constructor with these operations is called an **applicative** (functor)

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

Monads are applicatives

Every monad is also an applicative

```
pure = return
mf <*> mx = do f <- mf
               x <- mx
               return (f x)
```

But there are applicatives which are not monads!

As a result, you can use applicative style with `IO`, `[]`, `State`…

```
do x <- xs           ==  [ x + y
   y <- ys                   | x <- xs
   return (x + y)            , y <- ys ]

                     ==  (+) <$> xs <*> ys
```

The functor - applicative - monad hierarchy

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

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

class Applicative f => Monad f where
  -- return is the same as Applicative's pure
  (>>=) :: f a -> (a -> f b) -> f b
```

The functor - applicative - monad hierarchy

```
fmap        :: (a -> b)   -> f a -> f b
(<*>)       :: f (a -> b) -> f a -> f b
flip (>>=)  :: (a -> f b) -> f a -> f b
```

- `fmap` lifts a pure function, `(<*>)` has the function inside the type constructor
- With `(<*>)`, the outer context `f` is fixed, whereas with `(>>=)` this context depends on the value of `a`

  ```
  do x <- xs
     if x == 3
       then return 9
       else return 7
  ```

  is not expressible using only applicatives