

# Monads

## Advanced functional programming - Lecture 3

Wouter Swierstra and Alejandro Serrano



# In this lecture

- ▶ A number of useful programming patterns.
- ▶ We will see a similarity between seemingly different concepts.



# The Maybe type

```
data Maybe a = Nothing  
             | Just a
```

The Maybe datatype is often used to encode failure or an exceptional value:

```
find :: (a -> Bool) -> [a] -> Maybe a  
lookup :: Eq a => a -> [(a,b)] -> Maybe b
```



# Encoding exceptions using Maybe

Assume that we have a (Zipper-like) data structure with the following operations:

```
up, down, right :: Loc -> Maybe Loc  
update :: (Int -> Int) -> Loc -> Loc
```

Given a location 11, we want to move up, right, down, and update the resulting position with using `update (+1)` ...

Each of the steps can fail.



# Encoding exceptions using Maybe (contd.)

The straightforward implementation calls each function, checking the result before continuing.

```
case up 11 of
  Nothing -> Nothing
  Just 12 -> case right 12 of
    Nothing -> Nothing
    Just 13 -> case down 13 of
      Nothing -> Nothing
      Just 14 -> Just (update (+1) 14)
```



# Encoding exceptions using Maybe (contd.)

The straightforward implementation calls each function, checking the result before continuing.

```
case up 11 of
  Nothing -> Nothing
  Just 12 -> case right 12 of
    Nothing -> Nothing
    Just 13 -> case down 13 of
      Nothing -> Nothing
      Just 14 -> Just (update (+1) 14)
```

There's a lot of code duplication here!

Let's try to refactor out the common pattern.



# Refactoring

```
case up l1 of
  Nothing -> Nothing
  Just l2 -> case right l2 of
    Nothing -> Nothing
    Just l3 -> case down l3 of
      Nothing -> Nothing
      Just l4 -> Just (update (+1) l4)
```

We would like to:

- ▶ call a function that may fail;
- ▶ return `Nothing` when the call fails;
- ▶ continue somehow when the call succeeds.
- ▶ and lift a final result `update (+1) l4` into a `Maybe`.



# Capturing this pattern

We need to define an operator that takes two arguments:

- ▶ call a function that may fail:

Maybe a

- ▶ continue somehow when the call succeeds:

a -> Maybe b.





# Capturing this pattern

We need to define an operator that takes two arguments:

- ▶ call a function that may fail:

Maybe a

- ▶ continue somehow when the call succeeds:

a -> Maybe b.

```
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
f >>= g = case f of
  Nothing -> Nothing
  Just x   -> g x
```



# Returning results

Once we have computed the desired result, `update (+1) 14`, it is easy to turn it into a value of type `Maybe Loc`.

Although it's not very useful just yet, we can define the following function:

```
return :: a -> Maybe a  
return x = Just x
```



# Refactoring our code

```
case up l1 of
  Nothing -> Nothing
  Just l2 -> case right l2 of
    Nothing -> Nothing
    Just l3 -> case down l3 of
      Nothing -> Nothing
      Just l4 -> Just (update (+1) l4)
```



# Refactoring our code

```
up l1 >=> \l2 ->  
  case right l2 of  
    Nothing -> Nothing  
    Just l3 -> case down l3 of  
      Nothing -> Nothing  
      Just l4 -> Just (update (+1) l4)
```



# Refactoring our code

```
up l1 >>= \l2 ->  
right l2 >>= \l3 ->  
  case down l3 of  
    Nothing -> Nothing  
    Just l4 -> Just (update (+1) l4)
```



# Refactoring our code

```
up 11 >>= \12 ->  
right 12 >>= \13 ->  
down 13 >>= \14 ->  
Just (update (+1) 14)
```



# Refactoring our code

```
up l1 >>= \l2 ->  
right l2 >>= \l3 ->  
down l3 >>= \l4 ->  
return (update (+1) l4)
```



# Refactoring our code

```
up l1 >>= \l2 ->  
right l2 >>= \l3 ->  
down l3 >>= \l4 ->  
return (update (+1) l4)
```

We can simplify this even further to:

```
up l1 >>= right >>= down >>= return . update (+1)
```





# Imperative look-and-feel

Compare the following Haskell code:

```
up 11 >=> \12 ->  
right 12 >=> \13 ->  
down 13 >=> \14 ->  
return (update (+1) 14)
```

with this 'imperative' code:

```
12 := up 11;  
13 := right 12;  
14 := down 13;  
return (update (+1) 14);
```



# Imperative look-and-feel

In the imperative code, failure is an implicit side-effect;

In the Haskell version, we track the possibility of failure using `Maybe` and 'hide' the implementation with the sequencing operator.



# A variation: Either

Compare the datatypes

```
data Either a b = Left a | Right b
```

```
data Maybe a = Nothing | Just a
```

The datatype `Maybe` can encode exceptional function results (i.e., failure), but no information can be associated with `Nothing`. We cannot distinguish different kinds of errors.

Using `Either`, we can use `Left` to encode errors, and `Right` to encode successful results.



# Example

```
type Error = String
```

```
fac :: Int -> Either Error Int
```

```
fac 0 = Right 1
```

```
fac n | n > 0 = case fac (n - 1) of >  
                Left e  -> Left e  
                Right r -> Right (n * r)  
    | otherwise = Left "fac: negative argument"
```

Structure of sequencing looks similar to the sequencing for Maybe.



# Sequencing and returning for Either

We can define variations of the operations for Maybe:

```
(>>=) :: Either Error a ->  
  (a -> Either Error b) -> Either Error b  
f >>= g = case f of  
  Left e -> Left e  
  Right x -> g x
```

```
return :: a -> Either Error a  
return x = Right x
```



# Refactoring our fac function

The function can now be written as:

```
fac :: Int -> Either Error Int
fac 0 = return 1
fac n
  | n > 0 = fac (n - 1) >>= \r -> return (n * r)
  | otherwise = Left "fac: negative argument"
```



# Simulating exceptions

We can abstract completely from the definition of the underlying `Either` type if we define functions to throw and catch errors.

```
throwError :: Error -> Either Error a  
throwError e = Left e
```

```
catchError :: Either Error a ->  
              (Error -> a) ->  
              a  
catchError f handler = case f of  
    Left e -> handler e  
    Right x -> x
```



# State



Universiteit Utrecht

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



# Maintaining state explicitly

- ▶ We pass state to a function as an argument.
- ▶ The function modifies the state and produces it as a result.
- ▶ If the function does anything except modifying the state, we must return a tuple (or a special-purpose datatype with multiple fields).

This motivates the following type definition:

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



# Using state

There are many situations where maintaining state is useful:

- ▶ using a random number generator – like we saw for QuickCheck

```
type Random a = State StdGen a
```

- ▶ using a counter to generate unique labels

```
type Counter a = State Int a
```



# Using state – continued

- ▶ maintaining the complete current configuration of an application (an interpreter, a game, ...) using a user-defined datatype

```
data ProgramState = ...
```

```
type Program a = State ProgramState a
```

- ▶ caching information locally, which can later be flushed to an external data source, such as a database or file.



# Encoding state passing

```
data Tree a = Leaf a
            | Node (Tree a) (Tree a)

relabel :: Tree a -> State Int (Tree Int)
relabel (Leaf x)    = \s -> (Leaf s, s + 1)
relabel (Node l r) = \s ->
  let (l',s') = relabel l s
  in let (r',s'') = relabel r s'
  in (Node l' r', s'')
```

Again, we'll define two functions:

- ▶ a way to sequence the state from one call to the next;
- ▶ a way to produce a final results.



# Sequence and return for state

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

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



# Refactoring our code

```
relabel :: Tree a -> State Int (Tree Int)
relabel (Leaf x) = \s -> (Leaf s, s + 1)
relabel (Node l r) = \s ->
    let (l',s') = relabel l s
    in let (r',s'') = relabel r s'
    in (Node l' r', s'')
```

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

Let's try to refactor the code, using our sequencing operator.



# Refactoring our code

```
relabel :: Tree a -> State Int (Tree Int)
relabel (Leaf x)    = \s -> (Leaf s, s + 1)
relabel (Node l r) =
    relabel l >>= \l' -> \s' ->
    (r',s'') = relabel r s' in
    (Node l' r', s'')
```

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

Instead of threading the state explicitly, we can use >>=!



# Refactoring our code

```
relabel :: Tree a -> State Int (Tree Int)
relabel (Leaf x) = \s -> (Leaf s, s + 1)
relabel (Node l r) =
    relabel l >>= \l' ->
    relabel r >>= \r' -> \s'' ->
    (Node l' r', s'')
```

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

Now we observe that the final step is not modifying the state.





# Refactoring our code

```
relabel :: Tree a -> State Int (Tree Int)
relabel (Leaf x) = \s -> (Leaf s, s + 1)
relabel (Node l r) =
    relabel l >>= \l' ->
    relabel r >>= \r' ->
    return (Node l' r')

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



# Comparison with imperative version

In Haskell:

```
relabel l >>= \l' ->  
relabel r >>= \r' ->  
return (Node l' r')
```

Imperative pseudocode:

```
l' := relabel l;  
r' := relabel r;  
return (Node l' r');
```



# Comparison with imperative version

- ▶ In most imperative languages, the occurrence of memory updates is an implicit side effect.
- ▶ Haskell is more explicit because we use the State type and the appropriate sequencing operation.



# “Primitive” operations for state handling

We can completely hide the implementation of `State` if we provide the following two operations as an interface:

```
get :: State s s
get = \s -> (s, s)
```

```
put :: s -> State s ()
put s = \_ -> ((), s)
```

Using this we can define the following helper function for our example:

```
fresh :: State Int ()
fresh = get >=> \s -> put (s + 1)
```



# Haskell libraries

Actually, Haskell's `Control.Monad.State` module uses a slightly different implementation:

```
newtype State s a = State {runState :: s -> (a, s)}
```

This definition is equivalent to the definition we saw previously.



# Lists



Universiteit Utrecht

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

# Encoding multiple results and nondeterminism

Get the length of all words in a list of multi-line texts:

```
map length
  (concat
    (map words
      (concat (map lines txts)))))
```

► Easier to understand with a list comprehension:

```
[ length w | t <- txts, l <- lines t, w <- words l ]
```



# Sequencing again

We can also define sequencing and embedding, i.e., ( $>>=$ ) and return for lists:

```
(>>=) :: [a] -> (a -> [b]) -> [b]  
xs >>= f = concat (map f xs)
```

```
return :: a -> [a]  
return x = [x]
```





# Using bind and return for lists

Once again, we can refactor code to use bind, turning:

```
map length (concat (map words (concat (map lines txt
```

into:

```
txts >>= \t ->  
lines t >>= l ->  
words l >>= w ->  
return (length w)
```



# Comparison with imperative solution

- ▶ Again, we have a similarity to imperative code.
- ▶ In the imperative language, we have implicit nondeterminism (one or all of the options are chosen).
- ▶ In Haskell, we are explicit by using the list datatype and explicit sequencing using ( $>=>$ ).



# Intermediate Summary

At least three types with ( $\gg=$ ) and return:

- ▶ for Maybe, ( $\gg=$ ) sequences operations that may trigger exceptions and shortcuts evaluation once an exception is encountered; return embeds a function that never throws an exception;
- ▶ for State, ( $\gg=$ ) sequences operations that may modify some state and threads the state through the operations; return embeds a function that never modifies the state;
- ▶ for [], ( $\gg=$ ) sequences operations that may have multiple results and executes subsequent operations for each of the previous results; return embeds a function that only ever has one result.



There is a common interface here!

# The Monad class



# Monad class

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

- ▶ The name “monad” is borrowed from category theory.
- ▶ A monad is an algebraic structure similar to a monoid.
- ▶ Monads were first studied in the semantics of programming languages by Moggi; later they were applied to functional programming languages by Wadler.



# Instances

```
instance Monad Maybe where
```

```
...
```

```
instance (Error e) => Monad (Either e) where
```

```
...
```

```
instance Monad [] where
```

```
...
```

```
newtype State s a = State {runState :: s -> (a, s)}
```

```
instance Monad (State s) where
```

```
...
```



## Excursion: type constructors

- ▶ The class `Monad` ranges not over ordinary types, but over parameterized types.
- ▶ There are types of types, called *kinds*.
- ▶ Types of kind  $*$  are inhabited by values. Examples: `Bool`, `Int`, `Char`.
- ▶ Types of kind  $* \rightarrow *$  have one parameter of kind  $*$ . The `Monad` class ranges over such types. Examples: `[]`, `Maybe`.
- ▶ Applying a type constructor of kind  $* \rightarrow *$  to a type of kind  $*$  yields a type of kind  $*$ . Examples: `[Int]`, `Maybe Char`.
- ▶ The kind of `State` is  $* \rightarrow * \rightarrow *$ . For any type `s`, `State s` is of kind  $* \rightarrow *$  and can thus be an instance of class `Monad`.



# Excursion: functors

Monads are not the only 'higher-order' abstraction: structures that allow mapping have their own class.

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

- ▶ All containers, in particular all trees can be made an instance of functor.
- ▶ Every monad is a functor morally (`liftM`).
- ▶ Not all type constructors are functors; not all functors are monads...





# Monad laws

- ▶ Every instance of the monad class should have the following properties:
- ▶ `return` is the unit of `(>>=)`

```
return a >>= f == f a  
m >>= return == m
```

- ▶ associativity of `(>>=)`

```
(m >>= f) >>= g == m >>= (\x -> f x >>= g)
```



# Monad laws for Maybe

To prove the monad laws for `Maybe` we need to show for any  $f : a \rightarrow \text{Maybe } b$ , and for any  $m : \text{Maybe } a$ :

`Just x >>= f == f x`

and

`m >>= return == m`

Both are straightforward exercises.



# Monad laws for Maybe

To prove the monad laws for `Maybe` we need to show for any  $f : a \rightarrow \text{Maybe } b$ , and for any  $m : \text{Maybe } a$ :

`Just x >>= f == f x`

and

`m >>= return == m`

Both are straightforward exercises.

Similarly, associativity of `>>=` requires a longer, but no more complex proof.



# Bind or join

We have presented monads by defining the following interface:

```
(>>=) :: m a -> (a -> m b) -> m b  
return :: a -> m a
```

We could also have chosen the following, equivalent interface:

```
join    :: m (m a) -> m a  
return  :: a -> m a
```

It is a good exercise to try to define `>>=` in terms of `join` and visa versa (`m` also needs to be a functor).



# Monads are “monoids”



# Additional monad operations

Class `Monad` contains two additional methods, but with default methods:

```
class Monad m where
  ...
  (>>) :: m a -> m b -> m b
  m >> n = m >>= \_ -> n

  fail :: String -> m a
  fail s = error s
```

While the presence of `(>>)` can be justified for efficiency reasons, `fail` is used when desugaring the `do`-notation and should really be in a different class.



# do notation

Haskell offers special syntax for programming with monads.  
Rather than write:

```
mf >>= \f ->  
mg >>= \g ->  
...
```

You can also write:

```
do  
  f <- mf  
  g <- mg  
  ...
```

You can also use `let` expressions within `do` blocks to name  
(non monadic) computations.



# Monadic application

```
ap :: (Monad m) => m (a -> b) -> m a -> m b
ap mf mx = do
  f <- mf
  x <- mx
  return (f x)
```

Or without do notation:

```
ap mf mx = mf >>= \f' ->
  mx >>= \x' ->
  return (f x)
```





# More on do notation

- ▶ Use it, it is usually more concise.
- ▶ Never forget that it is just syntactic sugar. Use `(>>=)` and `(>>)` directly when it is more convenient.
- ▶ Remember that `return` is just a normal function:
  - ▶ Not every `do`-block ends with a `return`.
  - ▶ `return` can be used in the middle of a `do`-block, and it doesn't "jump" anywhere.
- ▶ Not every monad computation has to be in a `do`-block. In particular `do e` is the same as `e`.
- ▶ On the other hand, you may have to "repeat" the `do` in some places, for instance in the branches of an `if`.



# The IO monad

Another type with actions that require sequencing.

The IO monad is special in several ways:

- ▶ IO is a primitive type, and ( $\gg=$ ) and return for IO are primitive functions,
- ▶ there is no (politically correct) function  $\text{runIO} :: \text{IO } a \rightarrow a$ , whereas for most other monads there is a corresponding function,
- ▶ values of  $\text{IO } a$  denote side-effecting programs that can be executed by the run-time system.

Note that the specialty of IO has really not much to do with being a monad.



# IO, internally

```
> :i IO
newtype IO a
  = GHC.Types.IO
    (GHC.Prim.State# GHC.Prim.RealWorld
    -> (# GHC.Prim.State# GHC.Prim.RealWorld
        , a #))
    -- Defined in 'GHC.Types'
instance Monad IO -- Defined in 'GHC.Base'
...
```

Internally, GHC models IO as a state monad having the “real world” as state!



# The role of IO in Haskell

More and more features have been integrated into IO, for instance:

- ▶ classic file and terminal IO  
`putStr`, `hPutStr`
- ▶ references  
`newIORef`, `readIORef`, `writeIORef`
- ▶ access to the system  
`getArgs`, `getEnvironment`, `getClockTime`
- ▶ exceptions  
`throwIO`, `catch`
- ▶ concurrency  
`forkIO`



# I/O examples

Stdout output

```
> putStr "Hi"
```

Hi

```
> do putChar 'H' ; putChar 'i' ; putChar '!'
```

Hi!



# IO examples

## File IO

```
> do h <- openFile "TMP" WriteMode; hPutStrLn h "Hi"
> :q
Leaving GHCi
$ cat TMP
Hi
```



# IO examples

Side-effect: variables

```
do v <- newIORef "text"  
    modifyIORef v (\t -> t++ " and more text")  
    w <- readIORef v  
    print w
```

Results in

*text and more text*



# The role of IO in Haskell (contd.)

- ▶ Because of its special status, the IO monad provides a safe and convenient way to express all these constructs in Haskell. Haskell's purity (referential transparency) is not compromised, and equational reasoning can be used to reason about IO programs.
- ▶ A program that involves IO in its type can do everything. The absence of IO tells us a lot, but its presence does not allow us to judge what kind of IO is performed.
- ▶ It would be nice to have more fine-grained control on the effects a program performs.
- ▶ For some, but not all effects in IO, we can use or build specialized monads.





# Lifting functions to monads

```
liftM :: (Monad m) => (a -> b) -> m a -> m b  
liftM f m = do  x <- m; return (f x)
```

```
liftM2 :: (Monad m) => (a -> b -> c) -> m a -> m b -  
liftM2 f m1 m2 = do  x1 <- m1;  
                     x2 <- m2;  
                     return (f x1 x2)
```



# Lifting functions to monads

```
liftM :: (Monad m) => (a -> b) -> m a -> m b  
liftM f m = do  x <- m; return (f x)
```

```
liftM2 :: (Monad m) => (a -> b -> c) -> m a -> m b -  
liftM2 f m1 m2 = do  x1 <- m1;  
                     x2 <- m2;  
                     return (f x1 x2)
```

Question What is `liftM (+1) [1..5]`?



# Lifting functions to monads

```
liftM :: (Monad m) => (a -> b) -> m a -> m b  
liftM f m = do  x <- m; return (f x)
```

```
liftM2 :: (Monad m) => (a -> b -> c) -> m a -> m b -  
liftM2 f m1 m2 = do  x1 <- m1;  
                     x2 <- m2;  
                     return (f x1 x2)
```

Question What is `liftM (+1) [1..5]`?

Answer Same as `map (+1) [1..5]`. The function `liftM` generalizes `map` to arbitrary monads.



# Monadic map

```
mapM :: (Monad m) => (a -> m b) -> [a] -> m [b]
mapM f [] = return []
mapM f (x:xs) = liftM2 (:) (f x) (mapM f xs)
```

```
mapM_ :: (Monad m) => (a -> m b) -> [a] -> m () >
mapM_ f [] = return ()
mapM_ f (x:xs) = f x >> mapM_ f xs
```



# Sequencing monadic actions

```
sequence :: (Monad m) => [m a] -> m [a]  
sequence = foldr (liftM2(:)) (return [])
```

```
sequence_ :: (Monad m) => [m a] -> m ()  
sequence_ = foldr (>>) (return ())
```



# Monadic fold

```
foldM :: (Monad m) => (a -> b -> m a) -> a -> [b] ->  
foldM op e [] = return e  
foldM op e (x:xs) = do  
  r <- op e x  
  foldM f r xs
```



# More monadic operations

Browse `Control.Monad`:

```
filterM      :: (Monad m) => (a -> m Bool) -> [a] ->
replicateM   :: (Monad m) => Int -> m a -> m [a]
replicateM_  :: (Monad m) => Int -> m a -> m ()
join         :: (Monad m) => m (m a) -> m a
when         :: (Monad m) => Bool -> m () -> m ()
unless       :: (Monad m) => Bool -> m () -> m ()
forever      :: (Monad m) => m a -> m ()
```

...and more!



# Next lecture

- ▶ Next lecture continues with similar structures.
- ▶ Have a look at “Applicative Programming with Effects” by Conor McBride and Ross Paterson.

