Welcome to Programming Fundamentals 3

University of Luxembourg

2025/2026

Bachelor in Computer Sciences (BICS)
Semester 3

Lecture - Week #13

Dr. Afonso DELERUE ARRIAGA afonso.delerue@uni.lu

Important: Please remember to register your attendance in Moodle!

Today's lecture is about the state monad.

We will follow closely the recommended book for this course.

- Programming in Haskell, 2nd Edition, by Graham Hutton. September 2016, Cambridge University Press, ISBN-13 978-1316626221. Available via a-z.lu

Chapter 12, Section 12.3, Part "The state monad" and "Relabelling trees"

Recap from previous lecture

- In Haskell, data types can be made into functors, applicative functors and monads by declaring instances of typeclasses `Functor`, `Applicative` and `Monad`, respectively.
- This requires defining a few functions:
  - `fmap` for functors;
  - `pure` and `<*>` for applicatives;
  - `>>=` for monads.

Functors

- **Functors** allow mapping a function over each element of a data structure, while preserving the structure itself.

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

We can do things like:

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

inc = fmap (+1)

inc (Just 1)
inc [1,2,3,4,5]
inc (Node (Leaf 1) (Leaf 2))
```

Applicative functors

- **Applicatives** extend Functors by allowing to map functions with multiple arguments. Arguments are no longer plain arguments, but may in fact have effects such as *the possibility of failure* (`Maybe` or `Either`), *having many ways to succeed* (lists), or *performing IO actions* (`IO`).

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

We can do things like:

```
import Data.Char (toUpper)

pure (map toUpper) <*> getLine
Just (odd) <*> (Just (*3) <*> Just 5)
pure (+) <*> [1,2] <*> [3,4,5]
```

Monads

- **Monads** extend Applicative Functors by allowing composition of computations that depend on previous results.

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

We can turn this:

```
safeDiv :: Int -> Int -> Maybe Int
safeDiv _ 0 = Nothing
safeDiv m n = Just (m `div` n)

data Expr = Val Int | Div Expr Expr

eval :: Expr -> Maybe Int
eval (Val x) = Just x
eval (Div x y) = case eval x of
                     Nothing -> Nothing
                     Just m  -> case eval y of
                                    Nothing -> Nothing
                                    Just n  -> m `safeDiv` n
```

Into this:

```
eval :: Expr -> Maybe Int
eval (Val x) = return x
eval (Div x y) = eval x >>= (\m ->
                 eval y >>= (\n -> m `safeDiv` n))
```

Into this:

```
eval :: Expr -> Maybe Int
eval (Val x) = return x
eval (Div x y) = do m <- eval x
                    n <- eval y
                    safeDiv m n
```

Revisiting the bind operator `>>=`

- Why that sequence of bindings guarantees we never try to divide `Nothing`?

- First, let's recall how type `Maybe` is made into an instance of class `Monad`:

```
instance Monad Maybe where
    (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
    mx >>= f = case mx of
                   Nothing -> Nothing
                   Just x  -> f x
```

- Now let's unrol our monadic `eval` function:

```
data Expr = Val Int | Div Expr Expr

eval :: Expr -> Maybe Int
eval (Val x) = return x
eval (Div x y) = eval x >>= (\m ->
                 eval y >>= (\n -> m `safeDiv` n))
```

- Avoiding lambda functions, the previous form is equivalent to this:

```
eval :: Expr -> Maybe Int
eval (Val x) = return x
eval (Div x y) = eval x >>= f -- `eval x` has monadic type
`Maybe Int`
  where
    f :: (Int -> Maybe Int)
    f m = eval y >>= g
      where
        g :: (Int -> Maybe Int)
        g n = safeDiv m n
```

- Which is equivalent to this:

```
eval :: Expr -> Maybe Int
eval (Val x) = return x
eval (Div x y) = case eval x of
                     Nothing -> Nothing
                     Just x  -> eval y >>= g
  where
    g :: (Int -> Maybe Int)
    g n = safeDiv m n
```

- Which is turn is equivalent to the original form with multiple cases:

```
eval :: Expr -> Maybe Int
eval (Val x) = Just x
eval (Div x y) = case eval x of
                      Nothing -> Nothing
                      Just m  -> case eval y of
                                    Nothing -> Nothing
                                    Just n  -> m `safeDiv` n
```

- The `do` notation is just *syntactic sugar* (i.e. redundant syntax) for the bind operator `>>=`. Therefore, in the final version of `eval` with the `do` notation, the function `safeDiv` is only evaluated in case there's no previous failure.
- To conclude, we can sequence operations that may fail with the `Maybe` monad and its bind operator.

The list monad

- How to make lists into a monadic type?

```
instance Functor [] where
    fmap = map

instance Applicative [] where
    pure x = [x]
    gs <*> xs = [g x | g <- gs, x <- xs]
```

Exercise #1: Make `[]` an instance of class `Monad` 🧠

- Hint #1: What would be the type signature of `(>>=)` for `[]`?
- Hint #2: Work out the types.

In [ ]:
```
instance Monad [] where
    (>>=) :: [a] -> (a -> [b]) -> [b]
    xs >>= g = [y | x <- xs, y <- g x]
    --xs >>= g = concat $ map g xs
    --xs >>= g = concatMap g xs

[1,2,3] >>= (\x -> [x,x])
```

- We can view lists a bit like `Maybe` monad, but instead of carrying `Nothing` or `Just` a result, list might contain `0` (in the case of the empty list), `1` or multiple results.
- How to define the functions `pairs` using the fact that lists are monads?

```
pairs [1,2] [3,4]
[(1,3),(1,4),(2,3),(2,4)]
```

- Function `pair` in a monadic style:

```
pairs :: [a] -> [b] -> [(a,b)]
pairs xs ys = do
    x <- xs -- select x from list xs, in all possible ways
    y <- ys -- select y from list ys, in all possible ways
    return (x,y) -- combine
```

- This is quite similar to the list comprehension syntax:

```
pairs xs ys = [(x,y) | x <- xs, y <- ys]
```

The state monad

Introduction

- Consider the problem of writing functions that manipulate some form of state that can be changed over time.
- For simplicity, let's first consider that the state is simply an `Int`:

```
type InternalState = Int
```

- For instance, the state could represent a counter or the state of a finite-state machine.

- The most basic form of function on this type is a *state transformer*, abbreviated by `ST`.

```
type ST = InternalState -> InternalState
```

- The above type definition describes functions that consume a state and produce an updated state.

- In more general terms, we might want the state transformers to also output a result.

```
type ST a = InternalState -> (a, InternalState)
```

- We may also want that state transformers also take an argument as input. However, we don't need to further generalize the definition because of currying.
- We might be tempted to extend the definition to:

```
type InternalState = Int
type ST' a b = (a, InternalState) -> (b, InternalState)
```

- But any function of type `ST'` can be curried. For example:

```
import Data.Char

example :: ST' String String
example (x, state) = (map toUpper x, state + length x)

:t example
example :: ST' String String

:t curry example
curry example :: String -> InternalState -> (String,
InternalState)
```

- Notice that `curry example` is just a function that takes a `String` and outputs a `ST String`.
- Therefore, the previous type `ST` is general enough.

- Given that `ST` is a parameterised type, it is natural to try and make it into a monad so that the do notation can then be used to write stateful programs.
- However, types synonyms (declared via `type` mechanism) cannot be made into instances of classes.
- We have to use `newtype` or `data` declaration. For new types with a single constructor, `newtype` is prefered for efficiency reasons.

  ```
  newtype ST a = S (InternalState -> (a, InternalState))
  ```

- Notice that unfortunately, because `newtype` and `data` mechanisms require so, we were forced to introduce a dummy constructor `S`.

- Let's define a few helper functions that might be useful to deal with type `ST`:

  ```
  runState :: ST a -> InternalState -> (a, InternalState)
  runState (S f) s = f s

  evalState :: ST a -> InternalState -> a
  evalState st s = fst $ runState st s
  ```

- Function `runState` is simply an application function that allows to directly apply the state transformer function while avoiding having to pattern-match the dummy constructor `S`. In Graham Hutton's book, this function is called `app`.
- `evalState` outputs only the result.

Making the parameterised type ST into a monad

- Consider the new parameterized type `ST`.

  ```
  type InternalState = Int
  newtype ST a = S (InternalState -> (a, InternalState))
  ```

Exercise #2: Make it into a Functor, Applicative and Monad 🧠

- It's simply a puzzle of matching types...

In [ ]:
```haskell
type InternalState = Int
newtype ST a = S (InternalState -> (a, InternalState))

instance Functor ST where
    --fmap :: (a -> b) -> f a -> f b
    fmap :: (a -> b) -> ST a -> ST b
    fmap f (S g) = S h
      where
        -- f :: a -> b
        -- g :: InternalState -> (a, InternalState)
        h state = let (x, state') = g state
                  in  (f x, state')

instance Applicative ST where
```

```
    pure :: a -> ST a
    pure x = S (\state -> (x, state))

    (<*>) :: ST (a -> b) -> ST a -> ST b
    S f <*> S g = S h
      where
        -- f :: InternalState -> ((a -> b), InternalState)
        -- g :: InternalState -> (a, InternalState)
        -- h :: InternalState -> (b, InternalState)
        h state = let (f', state') = f state
                      (x, state'') = g state'
                  in  (f' x, state'')

instance Monad ST where
    (>>=) :: ST a -> (a -> ST b) -> ST b
    S f >>= g = S h
      where
        -- f :: InternalState -> (a, InternalState)
        -- g :: a -> ST b
        -- h :: InternalState -> (b, InternalState)
        h state = let (x, state') = f state
                  in  case g x of
                        S g' -> g' state'
```

A concrete example: relabelling trees 🌳

- As an example of stateful programming, let's write a function for relabelling function trees of the following type:

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

- Data type `Tree` is parameterized by generic type `a`. We could have, for example, a `Tree String`, where leafs are labelled with strings.

  ```
  t = Node (Node (Leaf "PF1") (Node (Leaf "PF2") (Leaf "PF3")))
  (Leaf "Software Foundations")
  ```

- We want to create a new function `relabel` that relabels a tree with unique `Int`. First, let's create the `relabel` function by passing around a counter.

Exercise #3: Relabelling trees passing around a counter 🎓

- Write a possible definition for a function `relabel`.

In [2]:
```
data Tree a = Leaf a | Node (Tree a) (Tree a) deriving Show

t = Node (Node (Leaf "PF1") (Node (Leaf "PF2") (Leaf "PF3"))) (Leaf "Software F

relabel :: Tree a -> Int -> (Tree Int, Int)
relabel (Leaf x) n = (Leaf n, n+1)
relabel (Node left right) n = (Node newLeft newRight, x)
  where
    (newLeft, n') = relabel left n
    (newRight, x) = relabel right n'
```

```
test1 :: Tree Int
test1 = fst $ relabel t 1

test1
```

```
Node (Node (Leaf 1) (Node (Leaf 2) (Leaf 3))) (Leaf 4)
```

Relabelling using the `ST` monad

- Now, let's try to use the `ST` monad we defined earlier.
- First, note that the signature of `relabel` resembles that of `ST`:

```
InternalState = Int
newtype ST a = S (InternalState -> (a, InternalState))

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

- We could rewrite `relabel` as using the type of state transformers:

```
relabel' :: Tree a -> ST (Tree Int)
```

- The state becomes the next *fresh* integer. Let's write an auxiliary function for that.

```
fresh :: ST Int
fresh = S (\x -> (x, x+1))
```

- How to rewrite the `relabel` function using `fresh` and the fact that `ST` is an Applicative Functor?

```
relabelA :: Tree a -> ST (Tree Int)
relabelA (Leaf x) = Leaf <$> fresh
relabelA (Node left right)= Node <$> relabelA left <*> relabelA
right

test2 :: Tree Int
test2 = evalState (relabelA t) 1
```

- In the base case we simply apply the `Leaf` constructor to the next *fresh integer*.
- In the recursive case we apply the `Node` constructor to the result of relabelling the two subtrees.

- We could also write this function in a monadic style:

```
relabelM :: Tree a -> ST (Tree Int)
relabelM (Leaf x) = do
    n <- fresh
    return (Leaf n)
relabelM (Node left right) = do
    left'  <- relabelM left
    right' <- relabelM right
    return (Node left' right')
```

```
test3 :: Tree Int
test3 = evalState (relabelM t) 1
```

- Both versions, `relabelA` and `relabelM` use `ST` monad to **abstract the state** from its definition.

Complementary material

- Checkout Graham Hutton's lecture on YouTube: AFP 9 - Monads III: State Revisited. 🏄
- Graham Hutton is the author of the book Programming in Haskell, the recommended companion for this course.

Haskell is my favorite imperative language ❤️

- Life is too short... How to use `Control.Monad.State` ?

```
import Control.Monad.State
-- % echo ":set -package mtl" >> ~/.ghci

--newtype ST a = S (s -> (a,s))
--type State s a = ST a

data Tree a = Leaf a | Node (Tree a) (Tree a) deriving Show

t :: Tree String
t = Node (Node (Leaf "PF1") (Node (Leaf "PF2") (Leaf "PF3")))
(Leaf "Software Foundations")

fresh :: State Int Int
fresh = do
    x <- get       -- get the current state
    put (x+1)      -- increment the state
--  modify (+1)    -- alternative formulation
    return x       -- return the current state

relabelM :: Tree a -> State Int (Tree Int)
relabelM (Leaf x) = do
    n <- fresh
    return (Leaf n)
relabelM (Node left right) = do
    left'  <- relabelM left
    right' <- relabelM right
    return (Node left' right')

test4 :: Tree Int
test4 = evalState (relabelM t) 1
```

Announcements for PF3

- Window to submit homework assignment closed on Sunday. We will grade it soon!
- Students that requested an early exam due to exceptional mobility constraints, please come to "TD" class on 17th December (no changes in classroom).
- For the other students, class on the 17th December is cancelled as announed on Moodle.
- The initial examination timetable indicates that the PF3 exam is scheduled for January 23rd. Students may bring an A4 sheet of paper with whatever notes they like. 📝
- Your feedback is important! Have your say on feedback.uni.lu. 🎓