Functional Programming
Lecture 12

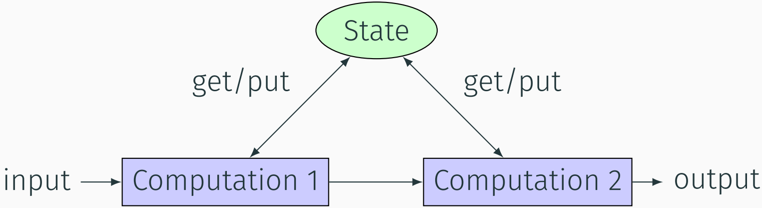
Rostislav Horčík

Czech Technical University in Prague
Faculty of Electrical Engineering
xhorcik@fel.cvut.cz

Stateful computations

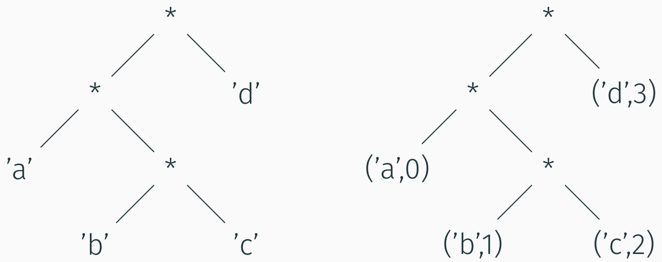
Stateful computation

Stateful computation uses a memory storage (state) to produce its output.



Tree labelling

Recall the exercise where we had to label tree leafs by consecutive natural numbers.



We need a state storing the information which numbers were already used.

Tree labelling

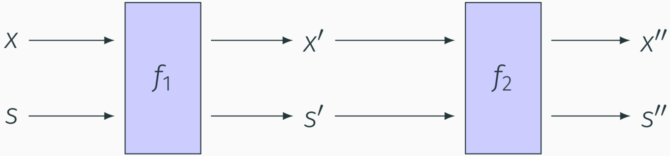
```haskell
data Tree a = Leaf a | Node (Tree a) (Tree a)
    deriving Show

labelHlp :: Tree a -> Int -> (Tree (a, Int), Int)
labelHlp (Leaf x) n = (Leaf (x, n), n+1)
labelHlp (Node left right) n =
    let (left', n') = labelHlp left n
        (right', n'') = labelHlp right n'
    in (Node left' right', n'')

labelTree :: Tree a -> Tree (a, Int)
labelTree t = fst (labelHlp t 0)
```

Stateful computation

In functional programming, we have to include state into function types.



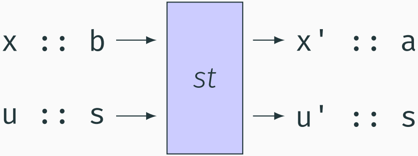
However, monads can help us to separate the state manipulation from the actual computation.

State monad

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

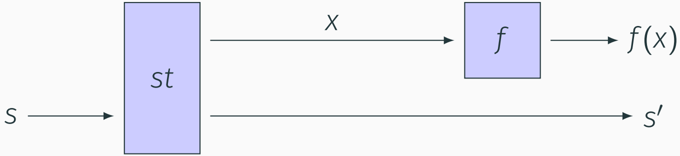
A stateful computation depending on a state of type `s` with an input of type `b` outputting a value of type `a`:

```haskell
st :: b -> State s a
```



Functor instance

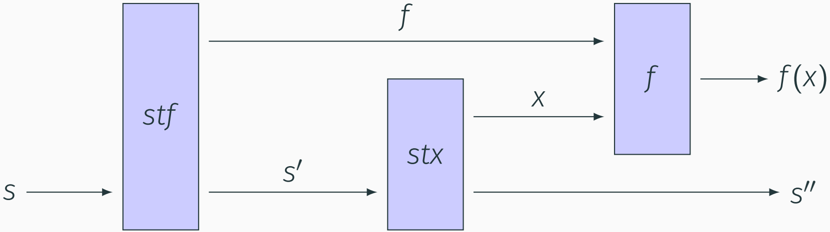
```
instance Functor (State s) where
-- fmap :: (a -> b) -> State s a -> State s b
  fmap f st = S (\s ->
    let (x,s') = runState st s
    in (f x,s'))
```



Applicative instance

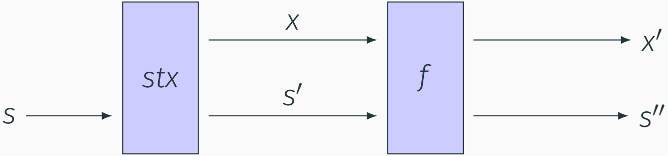
```haskell
instance Applicative (State s) where
-- pure :: a -> State s a
    pure x = S (\s -> (x,s))
-- (<*>) :: State s (a -> b) ->
--             State s a -> State s b
    stf <*> stx = S (\s ->
      let (f,s') = runState stf s
          (x,s'') = runState stx s'
      in (f x, s''))
```

Applicative instance



Monadic instance

```
instance Monad (State s) where
-- (>>=) :: State s a ->
--              (a -> State s b) -> State s b
   stx >>= f = S (\s ->
      let (x,s') = runState stx s
      in runState (f x) s')
```



Bind operator is just composition of stateful computations!

Functions manipulating state monad

State monad is actually implemented in `Control.Monad.Trans.State`. The library provides further useful functions.

```
state :: (s -> (a,s)) -> State s a
state f = S f

evalState :: State s a -> s -> a
evalState st x = fst $ runState st x

execState :: State s a -> s -> s
execState st x = snd $ runState st x
```

Tree labelling again

```haskell
fresh :: State Int Int
fresh = state (\n -> (n, n+1))

label :: Tree a -> State Int (Tree (a, Int))
label (Leaf x) = do i <- fresh
                    return $ Leaf (x, i)
label (Node l r) = do l' <- label l
                      r' <- label r
                      return $ Node l' r'

labelTree :: Tree a -> Tree (a, Int)
labelTree t = evalState (label t) 0
```

Functions manipulating state monad

Read, write and update of state can be done by

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

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

modify :: (s -> s) -> State s ()
modify f = do x <- get
              put (f x)
              return ()
```

Generating random values

Random values

A function returning a random value **cannot be pure** so it has to be enclosed inside `IO` monad.

However, we want most of our code to be pure.

Pseudorandom generators allow generating random values based on an initial seed.

$f(seed) = (x, newseed)$ where x is a random value

```haskell
rand100 :: Int -> (Int, Int)
rand100 seed = (n, newseed) where
  newseed = (1664525 * seed + 1013904223)
              `mod` (2^32)
  n = (newseed `mod` 100)
```

System.Random

Library `System.Random` is designed to generate pseudorandom values.

It uses values of `StdGen` as seed values (called generators). To create a new generator, call the function:

```
mkStdGen :: Int -> StdGen
```

Given a generator, a random value of type `a` in the given interval, can be generated by

```
randomR :: (RandomGen g, Random a) =>
  (a, a) -> g -> (a, g)
```

```
randomRIO :: Random a => (a, a) -> IO a
```

`Random` is a type class of the types for which we can generate pseudorandom values.

Generating a sequence

```
> randomR (0,100) (mkStdGen 1)
(46,80028 40692)
```

```
rand3Int :: Int -> StdGen -> ([Int], StdGen)
rand3Int m g0 = ([n1,n2,n3],g3)
    where
        (n1,g1) = randomR (0,m) g0
        (n2,g2) = randomR (0,m) g1
        (n3,g3) = randomR (0,m) g2
```

Pseudorandom values via state monad

```haskell
type R a = State StdGen a

randIntS :: Int -> R Int
randIntS m = state $ randomR (0,m)

rand3IntS :: Int -> R [Int]
rand3IntS n = do n1 <- randIntS n
                 n2 <- randIntS n
                 n3 <- randIntS n
                 return [n1,n2,n3]
```

Alternatively, we can use monadic version of `replicate`

```haskell
rand3IntS n = replicateM 3 (randIntS n)
```

Example

```haskell
manyRandIntsS :: Int -> R [Int]
manyRandIntsS n = mapM randIntsS $ repeat n

main :: IO ()
main = do
  seed <- randomIO :: IO Int
  putStrLn "How many random numbers do you want?"
  n <- read <$> getLine :: IO Int
  let rs = take n $ evalState
           (manyRandIntsS 100) (mkStdGen seed)
  print rs
```

Summary

- Stateful computations can be modelled via state monad.
- `State s a` encloses a function of type `s -> (a,s)`.
- It allows hiding of passing the state infomation.
- Pseudorandom values can be generated by functions from `System.Random`.
- State monad is useful to pass new generators.