Assignment 4 — Advanced Functional Programming, 2011/2012

Beerend Lauwers
Augusto Passalaqua
{B.Lauwers, A.PassalaquaMartins}@students.uu.nl

Utrecht University, The Netherlands

December 19, 2011

All exercises are defined in its own modules, imported by the lhs version of this document.

Exercise 1

```
{-# LANGUAGE ScopedTypeVariables #-}
module StackLanguage where
data Result = AResult [Int] deriving Show
store :: Result → Int → Result
(AResult xs) `store` x = AResult (x : xs)
mul :: Result → (Result → a) → a
mul (AResult (x : y : ys)) next = next $ AResult (x * y : ys)
add :: Result → (Result → a) → a
add (AResult (x : y : ys)) next = next $ AResult (x + y : ys)
stop = λ(AResult xs) → case xs of
  [] → error "No Integer on stack available!"
  otherwise → head xs
r &> x = store r x
l +> r = add l r
l *> r = mul l r
start = AResult []
p1 = start `store` 3 `store` 5 `add` stop
p2 = start `store` 3 `store` 6 `store` 2 `mul` add $ stop
p3 = start `store` 2 `add` stop
p1' = start &> 3 &> 5 +> stop
p2' = start &> 3 &> 6 &> 2 *> (+>stop)
p3' = start &> 2 +> stop
```

Exercise 2

```haskell
{-# LANGUAGE FlexibleInstances, MultiParamTypeClasses #-}
module Splittable where
import qualified System.Random as R
import Control.Monad.Reader
import Data.Map hiding (split)
class Splittable a where
  split :: a -> (a, a)
instance Splittable R.StdGen where
  split = R.split
instance (Splittable a) => Splittable [a] where
  split = organize ([], [])
    where
      organize (l, r) (x : xs) = let (n1, n2) = split x
        in organize (n1 : l, n2 : r) xs
      organize done _ = done
instance Splittable Int where
  split n = (2 * n, 2 * n + 1)
data SplitReader r a = SplitReader { runSplitReader :: r -> a }
  -- withSplitReader definition borrowed from Control.Monad.Reader.
withSplitReader :: (r' -> r) -> SplitReader r a -> SplitReader r' a
withSplitReader f m = SplitReader $ runSplitReader m . f
instance (Splittable r) => Monad (SplitReader r) where
  return a = withSplitReader split $ SplitReader (\r -> a)
  m >>= f = SplitReader (\r -> runSplitReader (f $ runSplitReader m (fst $ s r)) (snd $ s r))
    where s = split
type Bindings = [Int]
test :: SplitReader Bindings ()
test = return ()
instance (Splittable r) => MonadReader r (SplitReader r) where
  ask = SplitReader id
  local = withSplitReader
data Tree a = Leaf | Node (Tree a) a (Tree a) deriving Show
labelTree :: Int -> SplitReader Int (Tree Int)
labelTree 0 = return Leaf
labelTree n = return () >> liftM3 Node (labelTree (n - 1)) ask (labelTree (n - 1))
test2 :: SplitReader Int (Tree Int)
test2 = return () >> liftM3 Node (return Leaf) ask (return Leaf)
test3 :: SplitReader Int (Tree Int)
test3 = liftM3 Node (return Leaf) ask (return Leaf)
test4 :: SplitReader Int (Tree Int)
test4 = liftM3 Node (labelTree 1) ask (return Leaf)
test5 :: SplitReader Int (Tree Int)
test5 = liftM3 Node (labelTree 1) ask (labelTree 1)
```

Explanation

When removing the 'return () >>' clause, the result looks similar, only that the value has been split once more when recursing. This implies that the removed clause's side-effects (=the splitting of the value) are passed, even if the definition of (>>) notes as follows: "Sequentially compose two actions, discarding any value produced by the first (...)". Writing out the split values manually, however, it becomes clear that here it is not the case.

runSplitReader (labelTree 3) 0 results in the following values:

```
Node('
    Node('
        Node Leaf 2 Leaf'
        2'
        Node Leaf 26 Leaf'
        )'
    2'
    Node('
        Node Leaf 50 Leaf'
        26'
        Node Leaf 218 Leaf'
        )'
    )'
```

Replacing all the constructors and primitives with their split values, we get

```
(0,1)('
    (0,1)('
        (0,1) (0,1) (2,3) (6,7)'
        (2,3)'
        (6,7) (12,13) (26,27) (54,55)'
        )'
    (2,3)'
    (6,7)('
        (12,13) (24,25) (50,51) (102,103)'
        (26,27)'
        (54,55) (108,109) (218,219) (438,439)'
        )'
    )'
```

Semantics:

- Note that recursing reduces the produced value by 1. Hence, Nodes always pass the first element of the tuple.
- Every other element passes the second element of the tuple. (Remember that liftM3 rewrites the arguments in do-notation, which is how values are passed.)

- The passed value is used to generate a new tuple when the 'split' function is applied to it.
- ask returns the first element of the tuple.

Now, let's do the same for the original function definition:

```
return () >> Node('
                return () >> Node('
                                return () >> Node Leaf 86 Leaf'
                                             22'
                                             Node Leaf 374 Leaf'
                                )'
                            6'
                            Node('
                                return () >> Node Leaf 470 Leaf'
                                             118'
                                             Node Leaf 1910 Leaf'
                                )'
                )'
```

Replacing everything with their split values, we get:

```
(0,1) >> (0,1)('
                (2,3) >> (4,5)  ('
                                (10,11) >> (20,21) (42,43) \'
                                              (86,87) (174,175)'
                                           (22,23)'
                                           (92,93) (186,187) \'
                                              (374,375) (750,751)'
                               )'
                         (6,7)'
                         (28,29)('
                                (58,59) >> (116,117) (234,235) \'
                                             (470,471) (942,943)'
                                           (118,119)'
                                           (476,477) (954,955) \'
                                              (1910,1911) (3822,3823)'
                               )'
             )'
```

New semantics:

- Return () does the following:
  - Passes the first element of the tuple to the first argument of liftM3
  - Passes the second element of the tuple to the second argument of liftM3 (the 'ask' function)

- Node now passes its second element of the tuple and not its first.
- During passing of the 'ask' function's second element of the tuple, an extra side-effect occurs:
  - The element is split again, and the first element of this resulting tuple is passed to the third argument of liftM3. (In other words, the initial values is multiplied by 4)
- Note: The Node value of the third argument of liftM3 can also be calculated with the formula (second element of tuple of return())$*8+5$. ex: $59*8+4=476$.

Exercise 3

```
module PosParser where
```

This module uses the package EitherT in hackage.

```
import Control.Applicative
import Control.Monad.Trans.Either
import Control.Monad.Trans.Class
import Control.Monad.Trans.State
import Control.Monad.Trans.List
import Control.Monad.Identity
data PosParser a = PP ((Int, String) → Either Int [(a, (Int, String))])
instance Functor PosParser where
  f `fmap` PP p = PP (λinp → case p inp of
    (Right xs) → Right (map (λ(x, y) → (f x, y)) xs)
    (Left o) → Left o)
comb (Left p) (Left q)   = Left (p `max` q)
comb (Left _) (Right q) = Right q
comb (Right p) (Left _) = Right p
comb (Right p) (Right q) = Right (p ++ q)
instance Alternative PosParser where
  empty = PP (λ(pos, _) → Left pos)
  PP p <|> PP q = PP (λinp → (p inp) `comb` (q inp))
instance Applicative PosParser where
  PP p <*> ~ab@(PP q)
    = PP (λpi@(pos, inp) → case p pi of
      (Left pos) → Left pos
      (Right es) → foldr1 comb $ [test (r `fmap` ab) qi | (r, qi) ← es]
        )
  pure v = PP (λpi → Right [(v, pi)])
pSym v = PP (λ(pos, inp) → case inp of
  [] → Left pos
  (x : xs) → if x ≡ v
    then Right [(x, (pos + 1, xs))]
```

$\quad\quad\quad$ **else** *Left pos*

$\quad\quad\quad\quad$)

$\quad$ *test* (*PP p*) $i = p\ i$

$\quad$ $p1 = (\lambda a\ b \rightarrow a : [b]) < \$ > pSym$ `'a'` $< * > pSym$ `'b'`

$\quad$ **instance** *Monad PosParser* **where**

$\quad\quad$ *return* = *pure*

$\quad\quad$ $(PP\ a) \gg= b$

$\quad\quad\quad$ $= PP\ (\lambda inp \rightarrow$ **case** $a\ inp$ **of**

$\quad\quad\quad\quad$ $(Left\ pos) \rightarrow Left\ pos$

$\quad\quad\quad\quad$ $(Right\ es) \rightarrow foldr1\ comb\ \$\ [test\ (b\ r)\ qi \mid (r, qi) \leftarrow es]$

$\quad\quad\quad\quad\quad$)

Examples, working as expected:

$\quad$ *test p1* (0, `"ab"`)

Right [("ab",(2,""))]

$\quad$ *test p1* (0, `"abc"`)

Right [("ab",(2,"c"))]

$\quad$ *test p1* (0, `"ac"`)

Left 1

To get something isomorphic to the PosParser above we can use the monad transformers StateT, ListT and EitherT as follows:

$\quad$ **type** *PosParserIso a* = *StateT* (*Int*, *String*) (*ListT* (*EitherT Int Identity*)) *a*