Functional Programming (NWI-IBC040)

Resit — Monday, 6 April 2020 – 12:45 - 15:45

> Keep notice of the following: By taking this exam, the student declares that no plagiarism is or will be committed. If the lecturer has the suspicion that fraud has been committed, the student will be contacted. If needed, the case will be redirected to the Examination Board.

Please read carefully before answering the questions:

- Check that the exercise set is complete: the exam consists of 5 questions.
- The "cheat sheets' containing some Prelude functions is available as a separate document.
- Read the questions carefully.
- **Write your name and student number on *each* sheet you hand-in**.
- Write legibly. Be concise and precise.
- This exam is closed book. You are not allowed to use lecture notes, personal notes, books, a notebook, or any other electronic devices, except for a computer which is needed to register yourself via zoom and a mobile phone to scan and hand in your work. The use of these devices should be limited to this. It is not permitted to open GHCi or to search for solutions on the Internet.

1 Warm-up exercise (points: 2 × 3 = 6)

The function `maxDigit :: String → Int` returns the position of the maximum decimal digit in a string, with a result of -1 if the string contains no digits. The maximum digit appears more than once, `maxDigit` should return the first occurrence.

For example:

```
maxDigit "3 times 3 is 9" = 13
maxDigit "Digit 9 appears 9 times in 999999999" = 6
maxDigit "There is no digit in this sentence" = -1
maxDigit "7 is the maximum" = 0
```

1.a) Define `maxDigit` using basic/library functions and/or list comprehensions, but **not** recursion.

Solution:

```
maxDigit :: String → Int
maxDigit ds = snd $ foldr (\t r → if fst t ≥ fst r then t else r) (-1,-1)
                    [ (digitToInt d,i) | (d,i) ← zip ds [0..], isDigit d ]
```

or

```
maxDigit :: String → Int
maxDigit xs = fst ∘ maximumBy (comparing (\(x,y) → (y,Down x))) ∘ ((-1,'\0'):) ∘ filter
```

1.b) Now define `maxDigit`, but this time using recursion, and **not** list comprehension or any of the library functions for lists.

Solution:

```
maxDigit :: String → Int
maxDigit ds = md ds (-1,-1) 0 where
   md [] (_,p) _ = p
   md (d:ds) (m,p) i
      | isDigit d && digitToInt d > m = md ds (digitToInt d,i) (i+1)
      | otherwise                     = md ds (m,p) (i+1)
```

2 Functions and types (points: 8 × 2 = 16)

Define a *total* function for each of the following types, that is, provide a binding for each of the type signatures. You may assume that functions passed as arguments are total. Recall that `Either` is defined as `data Either a b = Left a | Right b`.

2.a) `f1 :: (a → a → b) → a → b`

Solution:

```
f1 x y = x y y
```

2.b) `f2 :: Either (a → b) b → a → b`

Solution:

```
f2 (Left f)  x = f x
f2 (Right y) _ = y
```

2.c) `f3 :: [a] → [b]`

Solution:

```
f3 _ = []
```

2.d) `f4 :: Applicative f ⇒ f a → f b → f (a,b)`

Solution:

```
f4 xs ys = (,) <$> xs <*> ys
-- $ ←- dollar sign to fix syntax highlighting in my text editor
```

Give the *most general type* for the following functions.

2.e) `g5 a = a >>= \x → return (x + 1)`

Solution:

```
g5 :: (Monad m, Num a) ⇒ m a → m a
```

2.f) `g6 = foldr (*) 1 ∘ map length`

Solution:

```
g6 :: [[a]] → Int
```

Note: recent versions of ghc will give the type

```
g6 :: Foldable f ⇒ [f a] → Int
```

Both are correct.

2.g) `g7 x y = filter (x ∘ head ∘ y)`

Solution:

```
g7 :: (b → Bool) → (a → [b]) → [a] → [a]
```

2.h) Consider a total function `unknown` with the type

```
unknown :: a → a → b → (b,a)
```

Give all possible values of the expression `unknown 2 3 "plus"`

Solution: From the generic type we know that the values of type `a` and `b` can only be passed to the output, and no computation can be performed. So the two possible outputs are

```
("plus",2)
("plus",3)
```

3 Trees (points: 6 × 3 = 18)

Consider the following datatype of polymorphic leaf trees, containing labels in the leaves .

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

3.a) Make `Tree` an instance of the type class `Functor`.

Solution:

```
instance Functor Tree where
   fmap f (Leaf e)   = Leaf (f e)
   fmap f (Node l r) = Node (fmap f l) (fmap f r)
```

A more general, monadic version of the fmap function has the following type:

```
mapM :: Monad m ⇒ (a → m b) → Tree a → m (Tree b)
```

Note that `mapM` has the same type as `fmap` for `Tree`, except that the argument function and the function itself now have monadic return types. The allows us to parameterize `mapM` with a function that produces a side-effect.

3.b) Implement `mapM`.

Solution:

```
mapM :: Monad m ⇒ (a → m b) → Tree a → m (Tree b)
mapM f (Leaf e)   = Leaf <$> f e
mapM f (Node l r) = Node <$> mapM f l <*> mapM f r
```

The simplest Monad we can think of is the Identity monad, which just wraps plain values, and does not incorporate any effect at all. The type for this monad is:

```
newtype IdM r = IdM r
```

3.c) Give appropriate instances of `IdM` for class `Functor` and for class `Monad`.

Solution:

```
instance Functor IdM where
   fmap f (IdM x) = IdM $ f x

instance Monad IdM where
   return =  IdM

   (IdM x) >>= g = g x
```

3.d) This monad can be used to give an alternative definition of `fmap` for `Tree`, expressed in terms of `mapM`. Again, make `Tree` an instance of the type class `Functor`, but now you should use `mapM` instead of recursion.

Solution:

```
instance Functor Tree where
  fmap f = fromIM ∘ mapM (return ∘ f)
```

Suppose we want to define a function `relabel :: Tree a → Tree Int` that relabels each leaf in such a tree with a unique or fresh integer. This could be implemented by taking the first fresh integer as an additional argument, and returning the next fresh integer as an additional result:

```
relabel' :: Tree a → Int → (Tree Int, Int)
```

However, instead of implementing `relabel` in this way you should use the `mapM`. This requires that you introduce a Monad that takes care of threading the argument containing the next fresh variable through your function. For this reason we introduce the state Monad, based on the following type:

```
newtype StateM r = StateM (Int → (r, Int))
```

It is convenient to define a special-purpose application function for this type, which simply removes the `StateM` constructor:

```
app :: StateM a → (Int → (a,Int))
app (StateM st) = st
```

3.e) Give appropriate instances for `StateM` of the class `Functor` and of the class `Monad`.

Solution:

```
instance Functor StateM where
  fmap f x = StateM $ \s → let (a,s') = app x s in (f a, s')
instance Monad StateM where
  return a = StateM $ \s → (a,s)
  x >>= y = StateM $ \s → let (a,s') = app x s in app y s'
```

3.f) Define `relabel` in terms of `mapM`.

Solution:

```
relabel tree = fst $ app (mapM label tree) 0
  where label _ = StateM $ \s → (s,s+1)
```

4 Fold/unfold (points: 5 × 3 = 15)

Recall the type of polymorphic leaf trees from the previous exercise.

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

The *base functor* (the base functor represents one layer of the type) for this type is

```
data TREE a t = LEAF a | NODE t t
```

Observe that this is a non-recursive data type.

4.a) Define an instance of class `Functor` for type `TREE`.

Solution:

```
instance Functor (TREE elem) where
  fmap f (LEAF a) = LEAF a
  fmap f (NODE l r) = NODE (f l) (f r)
```

The correspondence between the original recursive type and the base functor should be given as an instance of class Base:

```
class (Functor f) ⇒ Base f where
   type Rec f :: *
   inn  ::  f (Rec f) → Rec f
   out  ::  Rec f → f (Rec f)
```

4.b) Define the instance of class Base for type TREE.

Solution:

```
instance Base (TREE elem) where
  type Rec (TREE elem) = Tree elem

  inn (LEAF a) = Leaf a
  inn (NODE l r) = Node l r

  out (Leaf a) = LEAF a
  out (Node l r) = NODE l r
```

Use the higher-order operations

```
fold   :: (Base f) ⇒ (f a → a) → (Rec f → a)
unfold :: (Base f) ⇒ (a → f a) → (a → Rec f)
```

to implement the following functions. You *must not* use recursion. Remember that the type of fold for type Tree is

```
fold   :: (TREE a r → r) → (Tree a → r)
```

and for unfold the type is

```
unfold   :: (s → TREE a s) → (s → Tree a)
```

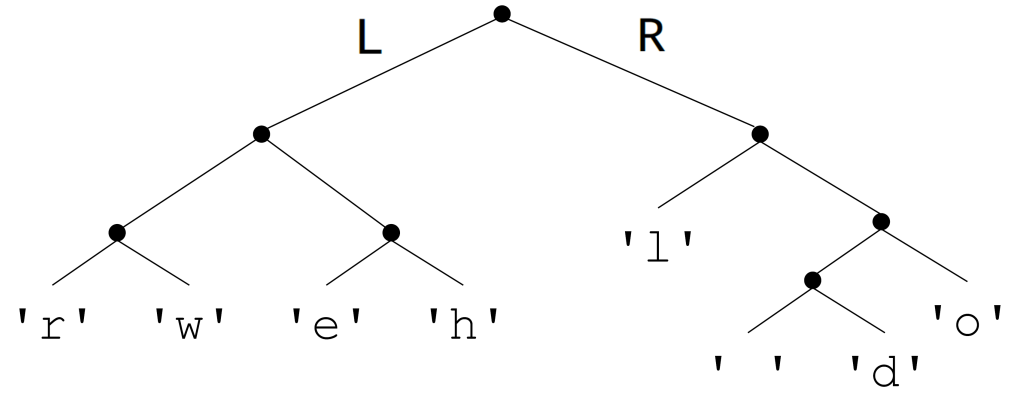
4.c) Define a function `toList` that collects all elements in a list, using `fold`.

```
toList :: Tree a → [a]
```

Solution:

```
toList = fold (\case LEAF el  → [el]
                     NODE l r → l ++ r)
```

Each element stored in a leaf-tree can be represented by its path, i.e. at each node of the tree a `L` and `R` indicate that the element is found in the left and right tree, respectively. Consider, for example the following leaf tree containing characters:



Starting at the root of this tree the code `LRR` guides us to `'h'` i.e. we walk left, right, and right again. Assume the following dat type declarations:

```
data Direction = L | R
type ElemTable a = [(a, [Direction])]
```

4.d) Use `fold` to define a function

```
treeToTable :: Tree a → ElemTable a
```

that converts a tree into an `ElemTable`.

For each element in the tree, this table should contain this element together with a the path from the root to it. *Hint*: At each node, all the paths occurring in the solution for the left and right subtrees should be prefixed with `L` and `R`, respectively.

If you are unable to write this function as a `fold`, you can define `treeToTable` differently. This will cost you a third of the points.

Solution:

```
treeToTable = fold step
  where
  step (LEAF x)   = [(x,[])]
  step (NODE l r) = [(x,O:path) | (x,path) ← l] ++ [(x,I:path) | (x,path) ←
r]
```

4.e) Use `unfold` to define a function

```
tableToTree :: ElemTable a → Tree a
```

that converts an element table to a corresponding leaf tree. *Hint*: At each node, the seeds for generating the left and right subtrees are the elements of the table containing paths that start with `L` and `R`, respectively.

Again, if you are unable to write this function as an `unfold`, you can define `tableToTree` differently. This will cost you a third of the points.

If necessary, you can assume that the table represents a complete tree and therefore will not lead to conversion errors.

Solution:

```
tableToTree = unfold step
  where
  step [(x,[])] = LEAF x
  step xs = NODE [(x,path) | (x,0:path) ← xs] [(x,path) | (x,I:path) ←
xs]
```

5 Correctness (points: 2 × 7 = 14)

For this exercise we extend the tree datatype from the previous exercises with a third constructor to represent empty trees:

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

along with these helper functions

```
fromList :: [a] → Tree a
fromList []     = Empty
fromList [x]    = Leaf x
fromList (x:xs) = Node (Leaf x) (fromList xs)

mapTree :: (a → b) → Tree a → Tree b
mapTree f Empty      = Empty
mapTree f (Leaf x)   = Leaf (f x)
mapTree f (Node x y) = Node (mapTree f x) (mapTree f y)

flipTree :: Tree a → Tree a
flipTree Empty      = Empty
flipTree (Leaf x)   = Leaf x
flipTree (Node x y) = Node (flipTree y) (flipTree x)
```

5.a) Prove by induction that

```
flipTree (flipTree t) = t
```

for all trees `t`.

Solution:

Answer:
Proof by induction on t.

Case 1: t = Empty.
flipTree (flipTree Empty)
 =
flipTree Empty
 =
Empty

Case 2: t = Leaf x.
flipTree (flipTree (Leaf x))
 =
flipTree (Leaf x)
 =
Leaf x

Case 3: t = Node x y.
Assume the induction hypotheses,
IH1: flipTree (flipTree x) = x
IH2: flipTree (flipTree y) = y

flipTree (flipTree (Node x y))
 =
flipTree (Node (flipTree y) (flipTree x))
 =
Node (flipTree (flipTree x)) (flipTree (flipTree y))
 = -- *IH*
Node x y

5.b) Prove by induction that for all f and xs

```
mapTree f (fromList xs) = fromList (map f xs)
```

Solution:

Answer:
Proof by induction on xs.

Case 1: xs = [].
mapTree f (fromList [])
 =
mapTree f Empty
 =
Empty
 =
fromList []

```
 =
fromList (map f [])

Case 2: xs = y:ys.
Assume the induction hypotheses,
IH: mapTree f (fromList ys) = fromList (map f ys)

case2a ys = []
mapTree f (fromList [y])
 =
mapTree f (Leaf y)
 =
Leaf (f y)
 =
fromList [f y]
 =
fromList (map f [y])
case2b ys /= []
mapTree f (fromList (y:ys))
 =
mapTree f (Node (Leaf y) (fromList ys))
 =
Node (mapTree f (Leaf y)) (mapTree f (fromList ys))
 = -- IH
Node (Leaf (f y)) (fromList (map f ys))
 =
fromList (f y : map f ys)
 =
fromList (map f (y : ys))
```