Functional Programming (NWI-IBC040)

Wednesday, 15 January 2020 – 12:45 - 15:45

Note: Please complete in block letters.

Surname:	....
First name:	....
Student number:	....

Please read carefully before answering the questions:

- Write your student number on *each* sheet.
- Check that the exercise set is complete: the exam consists of 5 questions.
- The last two pages of this exam are "cheat sheets" containing some Prelude functions.
- Read the questions carefully.
- Write your answers *on* the question sheets. Feel free to use the back pages, as well. Two blank pages are included at the rear. Additional paper is available on request.
- 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.

1 Warm-up exercise (6 points: 2 × 3)

The function addOddsOnEven :: [Int] → Int adds the odd numbers that are on even positions. For example:

```
addOddsOnEven [1,2,3,4,5,6] = 9
addOddsOnEven [] = 0
addOddsOnEven [3,5,7] = 10
addOddsOnEven [2,3,4,5,6] = 0
```

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

Solution:

```
addOddsOnEven xs = sum [ x | (x,p) ← zip xs [0..], odd x, even n ]
```

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

Solution:

```
addOddsOnEven []          = 0
addOddsOnEven [x]         = if odd x then x else 0
addOddsOnEven (x1:x2:xs) = if odd x then x + addOddsOnEven xs else addOddsOnEven xs
```

2 Higher-order polymorphism (16 points: 8 × 2)

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. Ad `g2`: Recall that `Either` is defined as `data Either a b = Left a | Right b`.

2.a) `g1 :: (a,b) → ((b, a),(a, b))`

Solution:

```
g1 (x,y) = ((y,x),(x,y))
```

2.b) `g2 :: (Either a () → a) → a`

Solution:

```
g2 f = f (Right ())
```

2.c) `g3 :: (((Bool → a) → a) → a) → a`

Solution:

```
g3 f = f (\g → g True)
```

2.d) `g4 :: (Functor f) ⇒ f (Maybe a) → f (Maybe (a,a))`

Solution:

```
g4 = fmap (fmap (\x → (x,x)))
```

2.e) `g5 :: (Functor f, Functor g) ⇒ g Char → f String → f (g String)`

Solution:

```
g5 gc fs =  fmap (\s → fmap (\c → c:s) gc) fs
```

2.f) `g6 :: (Monad m) ⇒ m (m ())`

Solution:

```
g6 = return (return ())
```

2.g) `g7 :: (Applicative f) ⇒ f a → f b → f (b,a)`

Solution:

```
g7 fa fb = pure (flip (,)) <*> fa <*> fb
```

2.h) `g8 :: (Monad m) ⇒ m (m (m a)) → m a`

Solution:

```
g8 mmm = mmm >>= \mm → mm >>= id
```

3 Polynomial functions (24 points: 8 × 3)

This assignment consists of several parts that can be made to a large extent independently of each other. So if you are unable to answer a question, proceed to the next one. If you need one of the preceding functions to answer a question, you can use it even if you have not been able to define it.

The following data type represents arithmetic expressions over a single variable, `X`. The type `Nat` is used to represent integer constants greater than or equal to zero. We assume that wherever a `Nat` value is used, it will always be non-negative.

```
type Nat = Int

data Expr
  = X
  | Lit Integer
  | Expr :+: Expr
  | Expr :*: Expr
  | Expr :^: Nat
  deriving (Eq, Show)
```

Note that the power operator (`:^:`) does not have an expression as a second parameter, but a natural number. This expression language represents *polynomial functions* of a single variable x. For example, the expression

```
expr1 :: Expr
expr1 = Lit 2 :+: (Lit 4 :*: X)  :+: (X :^: 3)
```

represents the function $f(x) = 2 + 4x + x^3$.

3.a) Write a function `evalExpr :: (Monad m) ⇒ Expr → m Integer` which given an expression returns the value of the expression. For now, the variable `X` evaluates to 0. As can be inferred from the type, this function should be written in *monadic style*.

```
evalExpr :: (Monad m) ⇒ Expr → m Integer
evalExpr X            =


evalExpr (Lit i)      =


evalExpr (e1 :+: e2)  =


evalExpr (e1 :*: e2)  =


evalExpr (e  :^: n)   =
```

Solution:

```
evalExpr :: (Monad m) ⇒ Expr → m Integer
evalExpr X           = 0
evalExpr (Lit i)     = return i
evalExpr (e1 :+: e2) = do v1 ← evalExpr e1; v2 ← evalExpr e2; return (v1 + v2)
evalExpr (e1 :*: e2) = do v1 ← evalExpr e1; v2 ← evalExpr e2; return (v1 + v2)
evalExpr (e :^: n)   = do v ← evalExpr e; return (v ^ n)
```

3.b) Now we ask you to extend your evaluator in such a way that it can be parameterized with a value for X. You should not add the value as an explicit argument, but use an appropriate monad that supports this form of parameter passing. Consequently, the type of the evaluator will no longer be parametric in the monad, i.e. it will change as follows:

```
evalExpr :: Expr → M Integer
```

for some concrete type constructor M.

Give an appropriate definition of M together with instances for class Functor and class Monad.

```
newtype         =

instance Functor        where
   fmap f =


instance Monad        where
   return x    =


   m >>= f     =
```

Solution:

```
newtype M a = M { fromM :: Integer → a }

instance Functor M where
   fmap f (M g) = M $ \env → f (g env)

instance Monad M where
   return = M ∘ const
   (M m) >>= f = M $ \env → fromM (f (m env)) env
```

Adjust the evaluator accordingly. You only have to give the function alternative that changes.

```
evalExpr
```

Solution:

```
ask :: M Integer
ask = M $ id

evalExpr X              = ask
```

The following data types represent polynomials in a single variable, `X`.

```
type Term = (Integer,Nat)
type Poly = [Term]
```

A polynomial consists of a sequence of *terms*. Each term is represented as a pair of coefficients and powers (of type `Integer` and `Nat`, respectively). We assume that terms are stored in increasing order of their exponents, and that all exponents of the terms are different. For instance, the polynomial

```
poly1 :: Poly
poly1 = [(2,0), (4,1), (1,3)]
```

represents the same function $f(x) = 2 + 4x + x^3$, as was given above.

3.c) Write a function `evalPoly :: Poly → Integer → Integer` which given a polynomial and the value of the variable `X` returns the value of the polynomial.

```
evalPoly :: Poly → Integer → Integer
```

Solution:

```
evalPoly :: Poly → Integer → Integer
evalPoly [] x         = 0
evalPoly ((c,e):ts) x = c * (x ^ e) + evalPoly ts x
```

3.d) Write a function `polyPlus :: Poly → Poly → Poly` that adds two polynomial together. `polyPlus` resembles the `zipWith` function in the sense that it traverses both lists simultaneously.

```
polyPlus :: Poly → Poly → Poly
```

Solution:

```
polyPlus :: Poly → Poly → Poly
polyPlus [] p2 = p2
polyPlus p1 [] = p1
polyPlus p1@((f1,e1):ts1) p2@((f2,e2):ts2)
   | e1 == e2  = (f1+f2, e1) : polyPlus ts1 ts2
   | e1 < e2   = (f1,e1)     : polyPlus ts1 p2
   | otherwise = (f2,e2)     : polyPlus p1 ts2
```

3.e) To multiply a polynomial with a term we introduce the function

```
termTimesPoly :: Term → Poly → Poly
```

which can be defined using `map`:

```
termTimesPoly (f1,e1) = map (\ (f2,e2) → (f1*f2,e1+e2))
```

Write a function `polyMul :: Poly → Poly → Poly` that multiplies two polynomials. The result of `polyMul p1 p2` can be obtained by first multiplying `p2` with each term of `p1` (e.g. by again using a map), and adding up the list of these intermediate results. The latter can be implemented using `foldr`

```
polyMul :: Poly → Poly → Poly
```

Solution:

```
polyMul :: Poly → Poly → Poly
polyMul p1 p2 = foldr polyPlus [] (map (flip termMul p2) p1)
```

3.f) Write a function `polyPow :: Poly → Nat → Poly` which given a polynomial p and an natural number n yields the polynomial p^n.

```
polyPow :: Poly → Nat → Poly
```

Solution:

```
polyPow :: Poly → Nat → Poly
polyPow p n = iterate (polyMul p) [(1,0)] !! n
```

3.g) Write a function `polyToExpr :: Poly → Expr` which converts a polynomial to the equivalent expression.

```
polyToExpr :: Poly → Expr
```

Solution:

```
polyToExpr :: Poly → Expr
polyToExpr ts = foldr (:+:) (Lit 0) (map (\(c,e) → Lit c :*: (X :^: e)) ts)
```

3.h) Write a function `exprToPoly :: Expr → Poly` which does the opposite, i.e. it converts an expression to the equivalent polynomial.

```
exprToPoly :: Expr → Poly
exprToPoly X           =
exprToPoly (Lit i)     =
exprToPoly (e1 :+: e2) =
exprToPoly (e1 :*: e2) =
exprToPoly (e  :^: n)  =
```

Solution:

```
exprToPoly X            = [(1,1)]
exprToPoly (Lit i)      = [(i,0)]
exprToPoly (e1 :+: e2)  = exprToPoly e1 `polyPlus` exprToPoly e2
exprToPoly (e1 :*: e2)  = exprToPoly e1 `polyMul`  exprToPoly e2
exprToPoly (e  :^: n)   = polyPow (exprToPoly e) n
```

4 Fold/unfold (18 points: 6 × 3)

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

```
data Tree elem = Single elem | Join (Tree elem) (Tree elem)
```

4.a) Define the base functor (the base functor represents one layer of the type) for this type together with the corresponding instance of class `Functor`.

```
data

instance Functor                  where
   fmap f
```

Solution:

```
data TREE elem res = SINGLE elem | JOIN res res

instance Functor (TREE elem) where
  fmap f (SINGLE a) = SINGLE a
  fmap f (JOIN l r) = JOIN (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 this class for (the base functor) of the tree type.

```
instance Base              where
```

Solution:

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

  inn (SINGLE a) = Single a
  inn (JOIN l r) = Join l r

  out (Single a) = SINGLE a
  out (Join l r) = JOIN 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.

4.c) Assume that we have a tree labelled with integers. Define a function `sum` that adds up all these integers, using `fold`.

```
sum :: Tree Integer → Integer
```

Solution:

```
sum :: Tree Integer → Integer
sum = fold (\case SINGLE x → x; JOIN l r → l + r)
```

4.d) Use `unfold` to define a function `grow` that creates a tree from a list of values by distributing the list elements over the binary tree such that the first half of elements will be stored in the left branch, and the second half in the right branch. If necessary, you may assume that this function is always called with a non-empty list.

```
grow :: [a] → Tree a
```

Solution:

```
grow :: [a] → Tree a
grow = unfold (\case [x] → SINGLE x
                     xs  → let h = length xs ‘div‘ 2
                               (l,r) = splitAt h xs
                           in JOIN l r)
```

4.e) Make `Tree` an instance of the type class `Functor`. Define `fmap` using `fold`.

Solution:

```
mapAsFold :: (a → b) → Tree a → Tree b
mapAsFold f = fold (\case SINGLE x → Single (f x); JOIN l r → Join l r)
```

4.f) Again, make `Tree` an instance of the type class `Functor`. Define `fmap` using `unfold`.

Solution:

```
mapAsUnfold :: (a → b) → Tree a → Tree b
mapAsUnfold f = unfold (\case Single x → SINGLE (f x); Join l r → JOIN l r)
```

5 Correctness (15 points: 4 + 3 + 8)

Consider the following function which inserts a value into a specific position in a list

```
insertAt :: Int → a → [a] → [a]
insertAt 0 x ys     = x : ys
insertAt _ _ []     = error "list too short"
insertAt i x (y:ys) = y : insertAt (i-1) x ys
```

For example

```
⋙ insertAt 2 'n' "lad" = "land"
```

5.a) `insertAt` gives an error if the position is larger than the length of the list. But when is this error raised? Which of the following expressions results in an error being shown on the screen when executed in GHCi?

(a) `take 5 (insertAt 10 'x' "example")`

(b) `length (insertAt 10 'x' "example")`

(c) `insertAt 7 'z' (insertAt 8 'x' "example")`

(d) `insertAt 8 'x' (insertAt 7 'z' "example")`

Solution: 2 and 3 result in an error

5.b) Implement a function `deleteAt` such that

```
deleteAt i (insertAt i x xs) = xs
```

for all $x, xs, i \leq$ `length` xs. Also give the type signature of this function. You do not have to prove this equivalence

```
deleteAt ::
```

Solution:

```
deleteAt :: Int → [a] → [a]
deleteAt i l = f ++ b where
  (f,c:b) = splitAt i l
```

5.c) Prove by induction that for all $x_1, x_2, xs, i_1 \leq i_2 \leq \mathtt{length}\ xs$:

$$\mathtt{insertAt}\ i_1\ x_1\ (\mathtt{insertAt}\ i_2\ x_2\ xs) = \mathtt{insertAt}\ (i_2 + 1)\ x_2\ (\mathtt{insertAt}\ i_1\ x_1\ xs)$$

Solution:

```
proof by induction on i_1

case 1: i_1 = 0

  insertAt i_1 x_1 (insertAt i_2 x_2 xs)
   =
  x_1 : (insertAt i_2 x_2 xs)
   =
  insertAt (i_2+1) x_2 (x_1 : xs)
   =
  insertAt (i_2+1) x_2 (insertAt i_1 x_1 xs)

case 2: i_1 > 0.
This implies that i_2>0
case 2a: xs=[]

  insertAt i_1 x_1 (insertAt i_2 x_2 [])
   =
  insertAt i_1 x_1 (error "list too short")
   =
  error "list too short"
   =
  insertAt (i_2+1) x_2 (error "list too short")
   =
  insertAt (i_2+1) x_2 (insertAt i_1 x_1 xs)

case 2b: xs=y:ys

  insertAt i_1 x_1 (insertAt i_2 x_2 (y:ys))
   =
  insertAt i_1 x_1 (y : insertAt (i_2-1) x_2 ys)
   =
  y : insertAt (i_1-1) x_1 (insertAt (i_2-1) x_2 ys)
   = {IH}
  y : insertAt i_2 x_2 (insertAt (i_1-1) x_1 ys)
   =
  insertAt (i_2+1) x_2 (y : insertAt (i_1-1) x_1 ys)
   =
  insertAt (i_2+1) x_2 (insertAt i_1 x_1 (y:ys))
```