Advanced Functional Programming Exam

DCC/FCUP — June 20th, 2015

- This exam comprises **five (5) questions** in **four (4) pages**
- You may use the provided Appendix sheet for Haskell library documentation
- Write your answers on an exam sheet; you may re-order your answers provided they are clearly marked

Question 1 (20%)

Recall the type systems presented in the course.

1. Find the principal type of the term $M \equiv (\lambda x.x(\lambda x.x)z)(\lambda f x.f x)$.
2. Using the Damas-Milner type system, infer a type for $\mathsf{let}\ f = (\lambda x.x)\ \mathsf{in}\ (f f)x$.
3. Consider the following class declaration:

   ```
   class a:C<=Eq where f: c -> c -> bool
   ```

   Which of the following types are valid instances of `C` (assume the existence of the projection function `fst` and of an equality class `Eq`)? Justify.

   ```
   inst int:C where f = \x->\y->x
   inst bool:C where f = \x->\y->y
   inst pair:(C,C)C where f = \x->\y->fst(x)==y
   ```

Question 2 (20%)

The `mapM` function from the `Control.Monad` library generalizes the standard list `map` for applying a monadic function over a list and collecting results; its most general type is:

```
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
```

Write a recursive definition of `mapM`; you may use do-notation or monadic `>>=` and `return` but *not* any other higher-order monadic functions.

Question 3 (20%)

Consider the following (erroneous) definition of a monad that keeps track of the number of applications of the bind operator `>>=`.

```
newtype M a = M { runM :: (a, Int) }

instance Monad M where
   return x = M (x,0)
   m >>= k = let (x,c) = runM m
                 (y,c')= runM (k x)
             in  M (y, 1+c+c')
```

Show that the monad laws do not hold for this definition, and hence `M` *is not* a monad. (Sugestion: it is enough to show that one of the laws fails).

Question 4 (20%)

Consider the following datatype for arithmetic expressions without variables:

```
data Expr = Const Integer
          | Add Expr Expr
          | Mul Expr Expr
          | Neg Expr
```

1. Write a QuickCheck generator for the `Expr` type, i.e. a function `genExpr :: Int -> Gen Expr` where integer argument is an upper-bound on the size of the result expression (for some suitable notion of "size").
2. Using `genExpr` make an instance of the `Arbitrary` type class for expressions.
3. Assume now that you are given an *evaluation function* `eval :: Expr -> Integer` for expressions. Write some QuickCheck properties for testing its correctness; you should write at least four distinct properties.
   (Sugestion: express algebraic properties such as $A + B = B + A$ or $A + (B + C) = (A + B) + C$.)

Question 5 (20%)

Consider a variant of the DSEL for turtle graphics presented in the lectures.

```
type Turtle a

instance Monad Turtle

forward :: Double -> Turtle ()   -- turtle movement
right :: Double -> Turtle ()     -- right rotation (degrees)
heading :: Turtle Double         -- get the current heading
position :: Turtle (Int,Int)     -- get the current position
distance :: Turtle Double        -- get total distance traveled
runTurtle :: Turtle a -> a       -- run function
```

The `distance` function should return the total distance traveled by the turtle; note that negative forward movement should also *increase* the distance traveled (e.g. as in a car's mileage meter)

The `runTurtle` function should execute the turtle commands and yield the result value. For example:

```
> runTurtle (forward 100 >> right 90 >> forward 100 >> position)
(100,-100)
> runTurtle (forward 100 >> forward (-50) >> distance)
150
```

Implement this DSL using a shallow embedding. Note that `runTurtle` should *not* perform any drawings (since its result type is not `IO`) — it should just simulate the turtle's behaviour.
