Exercises 3.1 and 3.2

F142001

June 19, 2015

Some preliminaries:

```
{-# LANGUAGE GADTs #-}
{-# LANGUAGE RankNTypes #-}

import Generics.LIGD
```

Exercise 1

```
data Tree a b = Tip a | Branch (Tree a b) b (Tree a b)
```

Kind: $* \to * \to *$
Explanation: one type for the inner nodes, and one type for the leafs. These types can be anything.

```
data GList f a = GNil | GCons a (GList f (f a))
```

Kind: $(* \to *) \to * \to *$
Explanation: The first argument must be a function that can take at least one argument (so it is of type $* \to *$). `Glist` then requires another argument.

```
data Bush a = Bush a (GList Bush (Bush a))
```

Kind: $* \to *$
Explanation: We only need to provide a single type (`a`), the rest is taken care of (i.e. the arguments of `GList` etc)

```
data HFix f a = HIn {hout :: f (HFix f) a}
```

Kind: $((* \to *) \to * \to *) \to * \to *$
Explanation: This data type has two parameters. The second one, `a`, can be anything. The first one, `f`, has to be a function that can take at least to arguments: something of type `HFix f` (i.e. $* \to *$), and something of type `a` (i.e. $*$).

```
data Exists b where
  Exists :: a -> (a -> b) -> Exists b
```

Kind: $* \to *$
Explanation: Only the single type `b` is needed as a parameter. The type `a` is also in there but it is not fixed as a parameter when we talk about the type `Exists b`; this type still allows for any `x :: A` and `f :: A -> b` in order to provide the proof for the existence of something of type `b` (i.e. `f(x)`).

```haskell
data Exp where
   Bool    :: Bool -> Exp
   Int     :: Int -> Exp
   GrT     :: Exp -> Exp -> Exp
   IsZero  :: Exp -> Exp
   Succ    :: Exp -> Exp
   If      :: Exp -> Exp -> Exp -> Exp
```

Kind: $*$
Explanation: There are no parameters to fill in.

Exercise 2

Exercise 2a

I had to change the definition from `GT` to `GrT` in order to avoid a conflict with `Prelude.GT`. Other than that, the code is pretty straightforward.

```haskell
eval :: Exp -> Maybe (Either Int Bool)
eval (Bool b)      = Just (Right b)
eval (Int  i)      = Just (Left i)
eval (GrT e1 e2)   = case eval e1 of
   Just (Left i1) -> case eval e2 of
      Just (Left i2)  -> Just (Right (i1 > i2))
      _ -> Nothing
   _ -> Nothing
eval (IsZero e)    = case eval e of
   Just (Left   i) -> Just (Right (i == 0))
   _               -> Nothing
eval (Succ e)      = case eval e of
   Just (Left   i) -> Just (Left (i+1))
   _               -> Nothing
eval (If c e1 e2) = case eval c of
   Just (Right True ) -> eval e1
   Just (Right False) -> eval e2
   _                  -> Nothing
```

Exercise 2b

`ExpF` should be of kind $* \to *$ (in order to become an argument to `Fix`). `r` is the recursion parameter.

```haskell
newtype Fix f = In {out :: f (Fix f)}

data ExpF r where
```

```haskell
  BoolF   :: Bool          -> ExpF r
  IntF    :: Int           -> ExpF r
  GTF     :: r -> r        -> ExpF r
  IsZeroF :: r             -> ExpF r
  SuccF   :: r             -> ExpF r
  IfF     :: r -> r -> r -> ExpF r

type Exp' = Fix ExpF
```

(Note that we could have written `data ExpF r = BoolF Bool | IntF Int | GTF r r | IsZeroF r | SuccF r | IfF r r r` if we wanted to avoid the GADT-notation. But I think this is more readable, especially in relation to the definitions in the other exercises.

Below, I give the isomorphism between `Exp'` and `Exp`. I omit the proof that this is actually an isomorphism.

```haskell
iso_to :: Exp' -> Exp
iso_to (In (BoolF b))     = Bool   b
iso_to (In (IntF  i))     = Int    i
iso_to (In (GTF x y))     = GrT    (iso_to x) (iso_to y)
iso_to (In (IsZeroF x))   = IsZero (iso_to x)
iso_to (In (SuccF x))     = Succ   (iso_to x)
iso_to (In (IfF c t e))   = If     (iso_to c) (iso_to t) (iso_to e)

iso_from :: Exp -> Exp'
iso_from (Bool b)     = In (BoolF b)
iso_from (Int  i)     = In (IntF  i)
iso_from (GrT x y)    = In (GTF (iso_from x) (iso_from y))
iso_from (IsZero x)   = In (IsZeroF (iso_from x))
iso_from (Succ x)     = In (SuccF (iso_from x))
iso_from (If c t e)   = In (IfF (iso_from c) (iso_from t) (iso_from e))
```

Exercise 2c

```haskell
instance Functor ExpF where
  fmap f (BoolF b)     = BoolF    b
  fmap f (IntF  i)     = IntF     i
  fmap f (GTF x y)     = GTF      (f x) (f y)
  fmap f (IsZeroF x)   = IsZeroF  (f x)
  fmap f (SuccF x)     = SuccF    (f x)
  fmap f (IfF c t e)   = IfF      (f c) (f t) (f e)
```

The following definition is given in the pdf file of the exercise:

```haskell
fold :: Functor f => (f a -> a) -> Fix f -> a
fold f = f . fmap (fold f) . out
```

From type signatures of `ExpF` and `fold`, we deduce the type signature of `evalAlg`

```haskell
evalAlg :: ExpF (Maybe (Either Int Bool)) -> Maybe (Either Int Bool)
evalAlg (BoolF   b) = Just (Right b)
evalAlg (IntF    i) = Just (Left  i)
evalAlg (GTF     (Just (Left i)) (Just (Left j))) = Just (Right (i > j))
evalAlg (GTF     _               _              ) = Nothing
evalAlg (IsZeroF (Just (Left i))) = Just (Right (i == 0))
evalAlg (IsZeroF _              ) = Nothing
evalAlg (SuccF   (Just (Left i))) = Just (Left (i + 1))
evalAlg (SuccF   _              ) = Nothing
evalAlg (IfF     (Just (Right True )) (Just t) _) = Just t
evalAlg (IfF     (Just (Right False)) _ (Just e)) = Just e
evalAlg (IfF     _                    _        _) = Nothing

eval' ::Exp' -> Maybe (Either Int Bool)
eval' = fold evalAlg
```

Here are some testing examples for `eval'`:

```haskell
example :: Exp'
example = In (IfF (In (BoolF False)) (In (IntF 5)) (In (IntF 6)))

example2 :: Exp'
example2 = In (SuccF (In (IntF 5)))

example3 :: Exp'
example3 = In (IfF (In (GTF (In (IntF 7)) (In (IntF 8)))) (In (SuccF (In (IntF 5))))
```

Exercise 2d

To become an argument to `HFix`, `ExpTF` should have kind $(* \to *) \to * \to *$

```haskell
data ExpTF r a where
  BoolTF    :: Bool                    -> ExpTF r Bool
  IntTF     :: Int                     -> ExpTF r Int
  GTTF      :: r Int -> r Int          -> ExpTF r Bool
  IsZeroTF  :: r Int                   -> ExpTF r Bool
  SuccTF    :: r Int                   -> ExpTF r Int
  IfTF      :: r Bool -> r a -> r a -> ExpTF r a

type ExpT' = HFix ExpTF
```

Example of an expression `e::Exp` that cannot be defined as an `ExpT'`:

```haskell
ex2d::Exp
ex2d = If (Bool True) (Bool True) (Int 1)
```

For the untyped version, we can simply evaluate this (the condition evaluates to true, so we pick the boolean value). For `ExpT'`, in order to define an `If`-expression, we need both arguments to be of the same type (not bool/int like in the example above).

Exercise 2e

The code below is similar to the previous cases, only with some extra (typing) parameters.

```
class HFunctor f where
  hfmap :: (forall b . g b -> h b) -> f g a -> f h a

instance HFunctor ExpTF where
  hfmap p (BoolTF b)     = BoolTF    b
  hfmap p (IntTF  i)     = IntTF     i
  hfmap p (GTTF x y)     = GTTF      (p x) (p y)
  hfmap p (IsZeroTF x)   = IsZeroTF  (p x)
  hfmap p (SuccTF   x)   = SuccTF    (p x)
  hfmap p (IfTF c t e)   = IfTF      (p c) (p t) (p e)

hfold :: HFunctor f => (forall b . f r b -> r b) -> HFix f a -> r a
hfold f = f.hfmap (hfold f) . hout

newtype Id a = Id {unId :: a}

evalAlgT :: ExpTF Id a -> Id a
evalAlgT (BoolTF b)               = Id b
evalAlgT (IntTF  i)               = Id i
evalAlgT (GTTF (Id i) (Id j))     = Id (i > j)
evalAlgT (IsZeroTF (Id i))        = Id (i == 0)
evalAlgT (SuccTF (Id i))          = Id (i+1)
evalAlgT (IfTF (Id True)  t e)    = t
evalAlgT (IfTF (Id False) t e)    = e

evalT' :: ExpT' a -> a
evalT' = unId . hfold evalAlgT
```

Test with: ("If 5 ¿ 38 then 10 else succ-10")

```
exampleH = HIn (IfTF (HIn (GTTF (HIn (IntTF 5)) (HIn (IntTF 38)))) (HIn (IntTF 10))
```

Exercises 3, 4 and 5 are to be found in the next document!