Dept. of Information and Computing Sciences, Utrecht University

**Generic Programming 2012**

# Solutions to Exercise Set 2

Sean Leather, Johan Jeuring

Tuesday, 18 September, 2012

## 1 General Information

Read the following instructions and notes.

### 1.1 Instructions

1. Read through all of the exercises before starting, so that you have an overall idea of what is expected and how much time to plan for each.
2. Create a file called `<First><Last>2.lhs` with `<First>` replaced by your first name (e.g. `Edsger`) and `<Last>` replaced by your surname (e.g. `Dijkstra`). Include your name and student number in comments.
3. Write your solution to each exercise in the file. Number the solutions in comments to match the exercise numbers.
4. Submit your file as an email attachment to leather@cs.uu.nl before the following deadline:

   **13:15 – Tuesday, 25 September, 2012**

### 1.2 Notes

- We recommend writing out answers by hand on paper before typing them. This will help you practice for the quizzes.
- You will need to install the latest `ligd` and `emgm` packages from Hackage.
- Note that the packages may use somewhat different names or notation from the AFP 2008 lecture notes.

- You may discuss the exercises amongst each other or with the lecturer at a conceptual level (in person, over IRC, or via email), but you cannot copy or share solutions. All work should be your own.
- Use the literate Haskell format for your submitted file. (Code follows > or goes between `\begin{code}` and `\end{code}` commands.) You don't need to do any other special formatting.
- Use GHC 7.4.*. GHC 7.4.1 comes with Haskell Platform 2012.2.0.0. GHC 7.6.1 is also available, but be aware that you may encounter issues if you use a version different from others.
- All code should type-check when the file is loaded into GHCi.
- The maximum possible score for the exercise set is 10. Next to each exercise number is its maximum possible score in parentheses.

Good luck!

## 2 Exercises

1. (2) If possible, give the LIGD representation of each type. If that is not possible, explain why not.

```
data Tree a b  = Tip a | Branch (Tree a b) b (Tree a b)
data GList f a = GNil | GCons a (f a)
data Bush a    = Bush a (GList Bush (Bush a))
data HFix f a  = HIn {hout :: f (HFix f) a}
data Exists b where
  Exists :: a → (a → b) → Exists b
data Exp where
  Bool   :: Bool              → Exp
  Int    :: Int               → Exp
  IsZero :: Exp               → Exp
  Add    :: Exp → Exp         → Exp
  If     :: Exp → Exp → Exp → Exp
```

**Solution.**

```
type Tree_S a b = a :+: Tree a b :×: b :×: Tree a b

fromTree :: Tree a b → Tree_S a b
fromTree (Tip x)            = L x
fromTree (Branch t1 x t2) = R (t1 :×: x :×: t2)

toTree :: Tree_S a b → Tree a b
toTree (L x)                  = Tip x
toTree (R (t1 :×: x :×: t2)) = Branch t1 x t2

rTree :: Rep a → Rep b → Rep (Tree a b)
rTree ra rb = RType (EP fromTree toTree)
                    (RSum (RCon "Tip" ra)
                          (RCon "Branch" (RProd (rTree ra rb)
                                                (RProd rb (rTree ra rb)))))
```

```
type GList_S f a = Unit :+: a :×: f a

fromGList :: GList f a → GList_S f a
fromGList GNil          = L Unit
fromGList (GCons x xs) = R (x :×: xs)

toGList :: GList_S f a → GList f a
toGList (L Unit)          = GNil
toGList (R (x :×: xs))    = GCons x xs

rGList :: (Rep a → Rep (f a)) → Rep a → Rep (GList f a)
rGList rf ra = RType (EP fromGList toGList)
                     (RSum (RCon "GNil" RUnit)
                           (RCon "GCons" (RProd ra (rf ra))))
```

```
type Bush_S a = a :×: GList Bush (Bush a)

fromBush :: Bush a → Bush_S a
fromBush (Bush x b) = x :×: b

toBush :: Bush_S a → Bush a
toBush (x :×: b) = Bush x b

rBush :: Rep a → Rep (Bush a)
rBush ra = RType (EP fromBush toBush)
                 (RCon "Bush" (RProd ra (rGList rBush (rBush ra))))
```

```
type HFixS f a = f (HFix f) a

fromHFix :: HFix f a → HFixS f a
fromHFix (HIn x) = x

toHFix :: HFixS f a → HFix f a
toHFix x = HIn x

rHFix :: ((Rep a → Rep (HFix f a)) → Rep a → Rep (HFixS f a))
      → Rep a
      → Rep (HFix f a)
rHFix rf ra = RType (EP fromHFix toHFix) (RCon "HIn" (rf (rHFix rf) ra))
```

We cannot define a representation for `Exists` because it is existential. We have no way of recovering the representation of the existentially quantified type `a`.

```
type ExpS = Bool :+: Int :+: Exp :+: Exp :×: Exp :+: Exp :×: Exp :×: Exp

fromExp :: Exp → ExpS
fromExp (Bool b)     = L b
fromExp (Int i)      = R (L i)
fromExp (IsZero e)   = R (R (L e))
fromExp (Add e1 e2)  = R (R (R (L (e1 :×: e2))))
fromExp (If c e1 e2) = R (R (R (R (c :×: e1 :×: e2))))

toExp :: ExpS → Exp
toExp (L b)                       = Bool b
toExp (R (L i))                   = Int i
toExp (R (R (L e)))               = IsZero e
toExp (R (R (R (L (e1 :×: e2))))) = Add e1 e2
toExp (R (R (R (R (c :×: e1 :×: e2))))) = If c e1 e2

rExp :: Rep Exp
rExp = RType (EP fromExp toExp)
             (RSum (RCon "Bool"   rBool)
             (RSum (RCon "Int"    RInt)
             (RSum (RCon "IsZero" rExp)
             (RSum (RCon "Add"    (RProd rExp rExp))
                   (RCon "If"     (RProd rExp (RProd rExp rExp))))))))

type BoolS = Unit :+: Unit

fromBool :: Bool → BoolS
fromBool False = L Unit
fromBool True  = R Unit

toBool :: BoolS → Bool
toBool (L Unit) = False
toBool (R Unit) = True

rBool :: Rep Bool
rBool = RType (EP fromBool toBool) (RSum RUnit RUnit)
```

2. (2) Define the generic function `typeInfo` in LIGD. The function should compute the sum of integers ( `Int` ), the maximum character ( `Char` ), and the list of constructors names (i.e. a value of type `[String]` ).

   **Solution.**

```
zero :: (Int, Char, [String])
zero = (0, minBound, [])

typeInfo :: Rep a → a → (Int, Char, [String])
typeInfo RChar c = (0, c, [])
typeInfo RInt i = (i, minBound, [])
typeInfo RString _ = zero
typeInfo RUnit _ = zero
typeInfo (RSum r _) (L x) = typeInfo r x
typeInfo (RSum _ r) (R y) = typeInfo r y
typeInfo (RProd r1 r2) (x :×: y) = case (typeInfo r1 x, typeInfo r2 y) of
  ((i1, c1, cs1), (i2, c2, cs2)) → (i1 + i2, max c1 c2, cs1 ++ cs2)
typeInfo (RCon con r) x = case typeInfo r x of
  (i, c, cs) → (i, c, con : cs)
typeInfo (RType ep r) x = typeInfo r (from ep x)
```

3. (2) Define a generic function in EMGM to compute the maximum "depth" of a value. Depth is measured by the number of nested constructors. For examples, `depth 'a' ≡ 0` , `depth [] ≡ 1` , `depth [1] ≡ 2` , `depth ["abc", "ab"] ≡ 5` , etc.

   **Solution.**

```
newtype Depth a = Depth {selDepth :: a → Int}

instance Generic Depth where
  runit       = Depth $ const 0
  rint        = Depth $ const 0
  rinteger    = Depth $ const 0
  rchar       = Depth $ const 0
  rfloat      = Depth $ const 0
  rdouble     = Depth $ const 0
  rsum ra rb  = Depth $ rsumDepth ra rb
  rprod ra rb = Depth $ rprodDepth ra rb
  rcon _ ra   = Depth $ succ ∘ selDepth ra
  rtype ep ra = Depth $ selDepth ra ∘ from ep

rsumDepth ra _ (L a) = selDepth ra a
rsumDepth _ rb (R b) = selDepth rb b

rprodDepth ra rb (a :×: b) = max (selDepth ra a) (selDepth rb b)

depth :: (Rep Depth a) ⇒ a → Int
depth = selDepth rep
```

4. (4) The following exercises use EMGM.

   a) (2) Give all the instances necessary to make the following types available for all generic functions in the library:

```
data Zig a b = ZigEnd a | ZigCons a (Zag a b)
data Zag a b = ZagEnd b | ZagCons b (Zig a b)
```

   **Solution.**

```
fromZig (ZigEnd a)      = L a
fromZig (ZigCons a z)   = R (a :×: z)

fromZag (ZagEnd b)      = L b
fromZag (ZagCons b z)   = R (b :×: z)

toZig (L a)             = ZigEnd a
toZig (R (a :×: z))     = ZigCons a z

toZag (L b)             = ZagEnd b
toZag (R (b :×: z))     = ZagCons b z

epZig  = (EP fromZig toZig)
epZag  = (EP fromZag toZag)
```

```
conZigEnd   = ConDescr "ZigEnd" 1 False Prefix
conZagEnd   = ConDescr "ZagEnd" 1 False Prefix

conZigCons  = ConDescr "ZigCons" 2 False Prefix
conZagCons  = ConDescr "ZagCons" 2 False Prefix
```

```
repZig ra rb =
  rtype epZig (rcon conZigEnd ra `rsum` rcon conZigCons (ra `rprod` repZag ra rb))
repZag ra rb =
  rtype epZag (rcon conZagEnd rb `rsum` rcon conZagCons (rb `rprod` repZig ra rb))
```

```
instance (Generic g, Rep g a, Rep g b) ⇒ Rep g (Zig a b) where
  rep = repZig rep rep
instance (Generic g, Rep g a, Rep g b) ⇒ Rep g (Zag a b) where
  rep = repZag rep rep
instance (Generic g, Rep g a) ⇒ FRep g (Zig a) where
  frep = repZig rep
instance (Generic g, Rep g a) ⇒ FRep g (Zag a) where
  frep = repZag rep
instance (BiFRep2 g Zag, Generic2 g) ⇒ BiFRep2 g Zig where
  bifrep2 ra rb =
    rtype2 epZig epZig
      (rcon2 conZigEnd ra `rsum2` rcon2 conZigCons (ra `rprod2` bifrep2 ra rb))
instance (BiFRep2 g Zig, Generic2 g) ⇒ BiFRep2 g Zag where
  bifrep2 ra rb =
    rtype2 epZag epZag
      (rcon2 conZagEnd rb `rsum2` rcon2 conZagCons (rb `rprod2` bifrep2 ra rb))
```

```
instance (Alternative f) ⇒ Rep (Collect f (Zig a b)) (Zig a b) where
  rep = Collect pure
```

```
instance (Rep (Everywhere (Zig a b)) a, Rep (Everywhere (Zig a b)) b)
         ⇒ Rep (Everywhere (Zig a b)) (Zig a b) where
  rep = Everywhere app
    where
      app f x =
        case x of
          ZigEnd a   → f (ZigEnd (selEverywhere rep f a))
          ZigCons a z →
            f (ZigCons (selEverywhere rep f a) (selEverywhere rep f z))
instance (Rep (Everywhere (Zag a b)) a, Rep (Everywhere (Zag a b)) b)
         ⇒ Rep (Everywhere (Zag a b)) (Zag a b) where
  rep = Everywhere app
    where
      app f x =
        case x of
          ZagEnd b   → f (ZagEnd (selEverywhere rep f b))
          ZagCons b z →
            f (ZagCons (selEverywhere rep f b) (selEverywhere rep f z))
```

```
instance Rep (Everywhere′ (Zig a b)) (Zig a b) where
  rep = Everywhere′ ($)
instance Rep (Everywhere′ (Zag a b)) (Zag a b) where
  rep = Everywhere′ ($)
```

b) (2) Define the following functions (and give their full type signatures) using EMGM. While some of these functions might be trivial to write by pattern-matching, imagine if the collection of datatypes here was much larger.

i. (0.5) `collectChars` collects a list of all characters from a `Zig` value:

```
collectChars :: (...) ⇒ Zig a b → [Char]
```

**Solution.**

```
collectChars
  :: (Rep (Collect [] Char) b, Rep (Collect [] Char) a) ⇒ Zig a b → [Char]
collectChars = collect
```

ii. (0.5) `collectZagChars` function collects a list of only the characters found in `Zag` values within a `Zig` value:

```
collectZagChars :: (...) ⇒ Zig a Char → [Char]
```

**Solution.**

```
collectZagChars :: (Rep (Crush [Char]) a) ⇒ Zig a Char → [Char]
collectZagChars = flattenr
```

iii. (0.5) `collectZigChars` collects a list of only the characters found in `Zig` values within a `Zig` value:

```
collectZigChars :: (...) ⇒ Zig Char b → [Char]
```

**Solution.**

```
newtype ZigRev b a = ToZigRev {fromZigRev :: Zig a b}
instance (Generic g, Rep g a) ⇒ FRep g (ZigRev a) where
  frep ra = rtype (EP fromZigRev ToZigRev) (repZig ra rep)
collectZigChars :: (Rep (Crush [Char]) b) ⇒ Zig Char b → [Char]
collectZigChars = flattenr ∘ ToZigRev
```

iv. (0.5) `collectZigZagChars` collects two lists of characters found in `Zig` values (without requiring the types to indicate). The first list is the collection of `Zig`-based characters, and the second is the collection of `Zag`-based characters. You may use `Data.Typeable.cast` here if you like.

```
collectZigZagChars :: (...) ⇒ Zig a b → ([Char], [Char])
```

**Solution.**

```
deriving instance Typeable2 Zig
collectZigZagChars
    :: (Rep (Crush [Char]) a, Rep (Crush [Char]) b, Typeable a, Typeable b)
   ⇒ Zig a b → ([Char], [Char])
collectZigZagChars z = (maybe [] collectZigChars  (castZig  z)
                                ,maybe [] collectZagChars (castZag z))
castZig :: (Typeable a, Typeable b) ⇒ Zig a b → Maybe (Zig Char b)
castZig = cast
castZag :: (Typeable b, Typeable a) ⇒ Zig a b → Maybe (Zig a Char)
castZag = cast
```