Dept. of Information and Computing Sciences, Utrecht University

Generic Programming 2012

Solutions to Exercise Set 1

Sean Leather, Johan Jeuring

Tuesday, 11 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>1.lhs` with `<First>` replaced by your first name (e.g. `Alonzo`) and `<Last>` replaced by your surname (e.g. `Church`). 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, 18 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` package from Hackage.
- You may discuss the exercises amongst each other or with the lecturers 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. (1.5) Consider each of the following Haskell datatypes.

```
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
```

   a) (0.5) What are the possible classifications of each datatype? (For example, an `Int` is both a primitive and a finite type.)

      Solution. The *italicized* term is required. The others are optional.

      - `Tree` : *regular*
      - `GList` : *higher-kinded*, regular, finite
      - `Bush` : *nested*
      - `HFix` : *higher-kinded*, nested
      - `Exists` : *existential*, GADT, finite
      - `Exp` : *regular*, not GADT even though it uses GADT syntax

b) (0.5) What is the kind of each datatype?

Solution.

$$
\begin{aligned}
&\mathsf{Tree} &&:: * \to * \to * \\
&\mathsf{GList} &&:: (* \to *) \to * \to * \\
&\mathsf{Bush} &&:: * \to * \\
&\mathsf{HFix} &&:: ((* \to *) \to * \to *) \to * \to * \\
&\mathsf{Exists} &&:: * \to * \\
&\mathsf{Exp} &&:: *
\end{aligned}
$$

c) (0.5) If possible, give the LIGD representation of each type. If not possible, explain why.

This solution will appear with the next exercise set.

2. (4.5) Use the Exp datatype above to do the following exercises.

a) (0.5) Write a function to interpret the Exp datatype above. Use the following type signature:

```
eval :: Exp → Maybe (Either Int Bool)
```

Note:

- IsZero expects an expression that evaluates to an Int and itself evalutes to True if the integer is 0 and False otherwise.
- Add takes two integer expressions and returns their sum.
- If takes one boolean expression and two other expressions of undetermined type. If the first argument evaluates to True, the second argument is returned. Otherwise, the third argument is returned.

Solution. This is one approach. Since Maybe is a Monad, it can also be written monadically.

```
eval (Bool b)     = Just (Right b)
eval (Int i)      = Just (Left i)
eval (IsZero e)   = case eval e of
                      Just (Left i) → Just (Right (i ≡ 0))
                      _             → Nothing
eval (Add e1 e2) = case eval e1 of
                      Just (Left i1) → case eval e2 of
                                         Just (Left i2) → Just (Left (i1 + i2))
                                         _              → Nothing
                      _              → Nothing
eval (If c e1 e2) = case eval c of
                     Just (Right b) → if b then eval e1 else eval e2
                     _              → Nothing
```

b) (0.5) Define a type `ExpF` such that `Exp'` is isomorphic to `Exp`.

```
newtype Fix f = In { out :: f (Fix f) }
type Exp' = Fix ExpF
```

Solution.

```
data ExpF :: * -> * where
  BoolF   :: Bool           -> ExpF r
  IntF    :: Int            -> ExpF r
  IsZeroF :: r              -> ExpF r
  AddF    :: r -> r         -> ExpF r
  IfF     :: r -> r -> r    -> ExpF r
```

c) (1) Give the `Functor` instance for `ExpF` and the evaluation algebra `evalAlg` such that for all isomorphic expressions `e :: Exp` and `e' :: Exp'`, `eval e ≡ eval' e'`.

```
fold :: Functor f => (f a -> a) -> Fix f -> a
fold f = f ∘ fmap (fold f) ∘ out
eval' :: Exp' -> Maybe (Either Int Bool)
eval' = fold evalAlg
```

Solution.

```
instance Functor ExpF where
  fmap f (BoolF b)      = BoolF b
  fmap f (IntF i)       = IntF i
  fmap f (IsZeroF e)    = IsZeroF (f e)
  fmap f (AddF e1 e2)   = AddF (f e1) (f e2)
  fmap f (IfF c e1 e2)  = IfF (f c) (f e1) (f e2)
evalAlg :: ExpF (Maybe (Either Int Bool)) -> Maybe (Either Int Bool)
evalAlg (BoolF b)     = Just (Right b)
evalAlg (IntF i)      = Just (Left i)
evalAlg (IsZeroF e)   = case e of
                          Just (Left i) -> Just (Right (i ≡ 0))
                          _             -> Nothing
evalAlg (AddF e1 e2)  = case e1 of
                          Just (Left i1) -> case e2 of
                                              Just (Left i2) -> Just (Left (i1 + i2))
                                              _              -> Nothing
                          _              -> Nothing
evalAlg (IfF c e1 e2) = case c of
                          Just (Right b) -> if b then e1 else e2
                          _              -> Nothing
```

d) (1) Define a GADT ExpTF such that $\mathsf{ExpT'}$ is well-typed (using type indexes) and isomorphic to $\mathsf{Exp'}$ if the extra types are erased.

```
type ExpT' = HFix ExpTF
```

Solution.

```
data ExpTF :: (* → *) → * → * where
  BoolTF   :: Bool                  → ExpTF r Bool
  IntTF    :: Int                   → ExpTF r Int
  IsZeroTF :: r Int                 → ExpTF r Bool
  AddTF    :: r Int → r Int         → ExpTF r Int
  IfTF     :: r Bool → r a → r a    → ExpTF r a
```

What is an expression $\mathsf{e :: Exp}$ that evaluates successfully (i.e. $\mathsf{eval\ e}$ does not result in $\mathsf{Nothing}$ or $\bot$) but cannot be defined in $\mathsf{ExpT'}$?

Solution. Something using If where the "true" and "false" terms have different types. Example:

```
e = If (Bool True) (Int 5) (Bool False)
```

e) (1.5) Study the code below carefully. Give the $\mathsf{HFunctor}$ instance for ExpTF and the evaluation algebra $\mathsf{evalAlgT}$ such that for all expressions $\mathsf{e' :: ExpT'}$ such that $\mathsf{evalT'\ e'}$ evaluates to a value v, the expression $\mathsf{eval\ e}$ in which is e is isomorphic to $\mathsf{e'}$ also evaluates to v.

```
class HFunctor f where
  hfmap :: (∀b . g b → h b) → f g a → f h a

hfold :: HFunctor f ⇒ (∀b . f r b → r b) → HFix f a → r a
hfold f = f.hfmap (hfold f) ∘ hout

newtype Id a = Id {unId :: a}

evalT' :: ExpT' a → a
evalT' = unId ∘ hfold evalAlgT

evalAlgT :: ExpTF Id a → Id a
```

Solution.

```haskell
instance HFunctor ExpTF where
  hfmap f (BoolTF b)      = BoolTF b
  hfmap f (IntTF i)       = IntTF i
  hfmap f (IsZeroTF e)    = IsZeroTF (f e)
  hfmap f (AddTF e1 e2)   = AddTF (f e1) (f e2)
  hfmap f (IfTF c e1 e2)  = IfTF (f c) (f e1) (f e2)

evalAlgT (BoolTF b)                         = Id b
evalAlgT (IntTF i)                          = Id i
evalAlgT (IsZeroTF (Id x))                  = Id (x ≡ 0)
evalAlgT (AddTF (Id i1) (Id i2))            = Id (i1 + i2)
evalAlgT (IfTF (Id c) (Id e1) (Id e2))      = Id (if c then e1 else e2)
```

3. (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]`).

   This solution will appear with the next exercise set.

4. (2) Define a type class `Desum` with an associated type `Desummed` and a function `desum`. The goal of `desum` is to take a value of a type `a` to a more general type, `Desummed a` , in which every use of `Either a b` is "flattened" to a pair `(Maybe a, Maybe b)` . Given instances for `()` , `Int` , `(a, b)` , and `Either a b` .

   Solution.

```haskell
class Desum a where
  type Desummed a
  desum :: a → Desummed a

instance Desum () where
  type Desummed () = ()
  desum = id

instance Desum Int where
  type Desummed Int = Int
  desum = id

instance (Desum a, Desum b) ⇒ Desum (a, b) where
  type Desummed (a, b) = (Desummed a, Desummed b)
  desum (x, y) = (desum x, desum y)

instance (Desum a, Desum b) ⇒ Desum (Either a b) where
  type Desummed (Either a b) = (Maybe (Desummed a), Maybe (Desummed b))
  desum (Left x)  = (Just (desum x), Nothing)
  desum (Right y) = (Nothing, Just (desum y))
```