Dept. of Information and Computing Sciences, Utrecht University

Theory of Programming and Types 2014

Solutions to Exercise Set 2

Johan Jeuring

February 2014

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 directory called <First><Last>2 with <First> replaced by your first name (e.g. Alonzo) and <Last> replaced by your surname (e.g. Church). The first questions ask you to write Haskell software. Answer these questions in a file with the extension lhs in this directory. The last question is about Agda, and answer this question in a file with the extension lagda.
3. Number your solutions in comments to match the exercise numbers.
4. Submit your file as an email attachment to J.T.Jeuring@uu.nl before the following deadline:

   13:15 – 13 March, 2014

1.2 Notes

- You will need to install the latest `ligd` and `regular` packages from Hackage.
- You may discuss the exercises amongst each other or with the lecturers at a conceptual level, 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.

- All code should type-check when the file is loaded into GHCi (Agda for the last question). You may use any version of GHC.
- 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. (0.5) 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
  GT     :: Exp → Exp         → Exp
  IsZero :: Exp               → Exp
  Add    :: Exp → Exp         → Exp
  If     :: Exp → Exp → Exp → Exp
```

What is the kind of each datatype?

Solution.

```
Tree   :: * → * → *
GList  :: (* → *) → * → *
Bush   :: * → *
HFix   :: ((* → *) → * → *) → * → *
Exists :: * → *
Exp    :: *
```

2. (2.5) Use the `Exp` datatype above to do the following exercises.

   a) 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.
- GT takes two integer expressions, and returns True if the first is greater than the second, 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 be written more elegantly monadically.

```
eval (Bool b)     = Just (Right b)
eval (Int i)      = Just (Left i)
eval (GT e1 e2)   = case eval e1 of
                      Just (Left i) → case eval e2 of
                                        Just (Left j) → Just (Right (i > j))
                                        _             → Nothing
                      _             → Nothing
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) 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
  GTF     :: r → r         → ExpF r
  IsZeroF :: r             → ExpF r
  AddF    :: r → r         → ExpF r
  IfF     :: r → r → r → ExpF r
```

c) 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 (GTF l r)      = GTF (f l) (f r)
   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 (GTF e1 e2)    = case e1 of
                           Just (Left i) → case e2 of
                                             Just (Left j) → Just (Right (i > j))
                                             _             → Nothing
                           _             → Nothing
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) Define a GADT ExpTF such that ExpT′ is well-typed (using type indexes) and isomorphic to 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
  GTTF     :: r Int → r Int       → ExpTF r Bool
  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 `e :: Exp` that evaluates successfully (i.e. `eval e` does not result in `Nothing` or `⊥`) but cannot be defined in `ExpT′`?

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

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

e) Study the code below carefully. Give the `HFunctor` instance for `ExpTF` and the evaluation algebra `evalAlgT` such that for all expressions `e′ :: ExpT′` such that `evalT′ e′` evaluates to a value `v`, the expression `eval e` in which is `e` is isomorphic to `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.

```
instance HFunctor ExpTF where
  hfmap f (BoolTF b)      = BoolTF b
  hfmap f (IntTF i)       = IntTF i
  hfmap f (GTTF l r)      = GTTF (f l) (f r)
  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 (GTTF (Id l) (Id r))       = Id (l > r)
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 a generic function using `regular` that collects the recursive children. The user-visible function is children, which is defined as:

```
children :: (R.Regular r, Children (R.PF r)) ⇒ r → [r]
children = children′ ∘ R.from
```

For example:

```
example3 = children [1,2] ≡ [[2]]
```

evaluates to True.

a) Define the Children type class with the single method children′.

Solution.

```
class Children f where
  children′ :: f r → [r]
```

b) Give instances of Children for the following functor types: unit, constant, constructor, recursive position, sum, and product.

Solution.

```
instance Children R.U where
  children′ R.U = []
instance Children (R.K a) where
  children′ (R.K _) = []
instance Children f ⇒ Children (R.C c f) where
  children′ (R.C x) = children′ x
instance Children R.I where
  children′ (R.I r) = [r]
instance (Children f, Children g) ⇒ Children (fR. :+: g) where
  children′ (R.L x) = children′ x
  children′ (R.R y) = children′ y
instance (Children f, Children g) ⇒ Children (fR. :×: g) where
  children′ (xR. :×: y) = children′ x ++ children′ y
```

4. (2) Define a generic function using `regular` that collects the subexpressions that are parents in a value of a datatype. A subexpression is a parent if it has a non-empty list of children. The user-visible function is parents, with the type:

```
parents :: (R.Regular r, ...) ⇒ r → [r]
```

For example:

```
example4 = parents [1,2,3] ≡ [[1,2,3],[2,3],[3]]
```

evaluates to `True`. Note that the subexpression `[]` is not among the parents, since it has no children.

Solution.

```
parents :: (R.Regular r, Children (R.PF r), Subelems (R.PF r)) ⇒ r → [r]
parents r = filter (¬ ∘ null ∘ children) (r : subelems r)

subelems :: (R.Regular r, Subelems (R.PF r)) ⇒ r → [r]
subelems = subelems′ ∘ R.from

class Subelems f where
  subelems′ :: (R.Regular r, Subelems (R.PF r)) ⇒ f r → [r]
```

Solution.

```
instance Subelems R.U where
  subelems′ R.U = []

instance Subelems (R.K a) where
  subelems′ (R.K _) = []

instance Subelems f ⇒ Subelems (R.C c f) where
  subelems′ (R.C x) = subelems′ x

instance Subelems R.I where
  subelems′ (R.I r) = r : subelems′ (R.from r)

instance (Subelems f, Subelems g) ⇒ Subelems (fR. :+: g) where
  subelems′ (R.L x) = subelems′ x
  subelems′ (R.R y) = subelems′ y

instance (Subelems f, Subelems g) ⇒ Subelems (fR. :×: g) where
  subelems′ (xR. :×: y) = subelems′ x ++ subelems′ y
```

```
type instance (R.PF) [a] = R.U R.:+: (R.K a R.:×: R.I)
instance R.Regular [a] where
  from [] = R.L R.U
  from (x : xs) = R.R (R.K x R.:×: R.I xs)
  to (R.L R.U) = []
  to (R.R (R.K x R.:×: R.I xs)) = x : xs
example4 = subelems [1, 2, 3, 4]
type instance R.PF (Tree a b) = R.K a R.:+: (R.I R.:×: (R.K b R.:×: R.I))
instance R.Regular (Tree a b) where
  from (Tip x) = R.L (R.K x)
  from (Branch l n r) = R.R (R.I l R.:×: (R.K n R.:×: R.I r))
  to (R.L (R.K x)) = Tip x
  to (R.R (R.I l R.:×: (R.K n R.:×: R.I r))) = Branch l n r
deriving instance (Show a, Show b) ⇒ Show (Tree a b)
example5 = subelems (Branch (Branch (Tip 0) 'a' (Tip 1)) 'b' (Tip 2))
```

5. (3) Implement the embedding from Regular into MultiRec in José Pedro Magalhães framework for formally proving embeddings of generic programming libraries. Most of the code can be found here: `http://www.dreixel.net/research/code/fcadgp.agda`. To check this code you need version 0.7 of the Agda library, available here: `http://www.cse.chalmers.se/~nad/software/lib-0.7.tar.gz`. Implement a module Regular2Multirec.