Dept. of Information and Computing Sciences, Utrecht University

Generic Programming 2012

Solutions to the Final Exercise Set

Sean Leather, Johan Jeuring

Sunday, 04 November, 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 exercise.
2. Create a file called `<First><Last>4.lhs` with `<First>` replaced by your first name (e.g. `Alan`) and `<Last>` replaced by your surname (e.g. `Turing`). 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:

   23:59 – Sunday, 11 November, 2012

1.2 Notes

- This exercise set contains exercises developed by your fellow classmates. It is not allowed to ask the developer of the exercise for hints.
- You may discuss the exercises amongst each other or with the lecturers at a conceptual level (in person, or via email), but you cannot copy or share solutions. All work should be your own.
- Direct your questions to both lecturers, in case one of them is not available to respond.

- 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. (3) For this question, refer to the paper "Uniform Boilerplate and List Processing" and/or the presentation by Jaap van der Plas. Use the "uniplate" package.

   Consider the following datatype:

```
data Expr
  = Add Expr Expr
  | Let String Expr Expr
  | Val Int
  | Var String
```

   a) (0.5) Define an instance of `Expr` for the `Uniplate` class by generating it (e.g. using the tool and Hackage package "derive") or by writing it yourself (e.g. adapting the instance of the similar datatype in the paper).

      Solution. This is one possible instance:

```
instance Uniplate Expr where
  uniplate (Add e1 e2)   = plate Add |* e1 |* e2
  uniplate (Let s e1 e2) = plate Let |- s |* e1 |* e2
  uniplate x             = plate x
```

   b) (1.5) Define a function –

```
removeUnusedBinds :: Expr -> Expr
```

      – that removes unused binds from an expression. An unused bind is a (non-recursive) `Let` whose variable name (first argument) is not referenced by a `Var` in the body (third argument). For example, the expression (in concrete syntax) –

```
1 + (let x = 1 in 2)
```

– should be transformed into the expression:

```
1 + 2
```

All operations that query or transform should be done with generic functions.

Solution. This is one possible solution, using `transform` and `universe`:

```
removeUnusedBinds e = transform f e
  where
    f (Let v e1 e2) | isUsed v e2 = Let v e1 e2
                    | otherwise   = e2
    f e'                          = e'
isUsed :: String → Expr → Bool
isUsed v e = any (≡ v) [v' | Var v' ← universe e]
```

c) (1) (Only attempt this part after successfully implementing part 1b.) Be sure that `removeUnusedBinds` properly handles shadowed bindings. The expression –

```
let x = 3 in (let x = 1 in 2 + x)
```

– should be transformed into:

```
let x = 1 in 2 + x
```

Note that `x` is shadowed by the inner binding, so the outer binding of `x` can be removed. Rewrite `removeUnusedBinds` if necessary.

Solution. The outer part of this solution is mostly unchanged from before. Below are two possible definitions of a function to determine if a variable is free in an expression.

```haskell
removeUnusedBinds e = transform f e
  where
    f (Let v e1 e2) | isFree v e2 = Let v e1 e2
                    | otherwise   = e2
    f e'                          = e'

isFree :: String → Expr → Bool
isFree v = go
  where
    go :: Expr → Bool
    go (Let v' e1 e2) | v ≡ v' = go e1
    go (Var v')       | v ≡ v' = True
    go e'                      = or (map go (children e'))

isFree :: String → Expr → Bool
isFree v = para f
  where
    f :: Expr → [Bool] → Bool
    f (Let v' e1 e2) | v ≡ v' = head
    f (Var v')       | v ≡ v' = const True
    f e'                      = or
```

2. (4) For this question, refer to the paper "Primitive (Co)Recursion and Course-of-Value (Co)Iteration, Categorically" and/or the presentation by João Alpuim.

   For each of the following functions –

   a) catamorphism (`cata`)
   b) anamorphism (`ana`)
   c) paramorphism (`para`)
   d) apomorphism (`apo`)

   – do the following:

   a) Give a brief description of the function and its relationship to the functions before it in the list (e.g. how anamorphism is related to catamorphism, not vice versa).
   b) Define the function using the following incomplete definitions:

```haskell
newtype Fix f = In {out :: f (Fix f)}
(&&&) :: (c → a) → (c → b) → c → (a, b)
(|||) :: (a → c) → (b → c) → Either a b → c
```

   c) Give an example of the function's use, including any datatypes and type class instances needed.

Note that due to the flexible nature of the notation used in this paper, the notation will not always match the function types you expect. For example, the combinators are used for both functor and non-functor types. Feel free to define the examples in a different way than given in the paper.

Solution.

We first define the basic combinators:

```
f &&& g = λx → (f x, g x)
f ||| g = λx → case x of
  Left y → f y
  Right y → g y
```

For lists, we use the following:

```
data ListF a r = NilF | ConsF a r
type List a = Fix (ListF a)
instance Functor (ListF a) where
  fmap _ NilF          = NilF
  fmap f  (ConsF a r) = ConsF a (f r)
```

```
data NatF r = ZeroF | SuccF r
type Nat = Fix NatF
instance Functor NatF where
  fmap _ ZeroF      = ZeroF
  fmap f  (SuccF r) = SuccF (f r)
zero'    = In ZeroF
succ' n = In (SuccF n)
```

The *catamorphism* is the natural recursion scheme of induction on an algebraic datatype. It is also called iteration.

```
cata :: Functor f ⇒ (f a → a) → Fix f → a
cata f = f ∘ fmap (cata f) ∘ out
(| − |) :: c → (a → b → c) → ListF a b → c
f | − | g = λx → case x of
  NilF        → f
  ConsF a r → g a r
sum' :: List Integer → Integer
sum' = cata (0 | − | (+))
```

The *anamorphism* is the natural corecursion scheme of coinduction on an coalgebraic codatatype. It is also called coiteration. It is the dual of the catamorphism.

```
ana :: Functor f ⇒ (a → f a) → a → Fix f
ana f = In ∘ fmap (ana f) ∘ f

(><) :: (a → c) → (b → d) → (a, b) → (c, d)
f >< g = (f ∘ fst) &&& (g ∘ snd)

zip′ :: (Fix ((,) c), Fix ((,) d)) → Fix ((,) (c, d))
zip′ = ana (((fst >< fst) &&& (snd >< snd)) ∘ (out >< out))
```

The *paramorphism* is the primitive recursion scheme. It allows functions to "eat the argument and keep it too." It is a generalization of the catamorphism.

```
para :: Functor f ⇒ (f (a, Fix f) → a) → Fix f → a
para f = f ∘ fmap (para f &&& id) ∘ out
  -- Alternative
para :: Functor f ⇒ (f (a, Fix f) → a) → Fix f → a
para f = fst ∘ cata (f &&& (In ∘ fmap snd))

mult :: (Nat, Nat) → Nat
mult (In ZeroF, n)     = In ZeroF
mult (In (SuccF r), n) = mult (r, n)

(|+|) :: c → (b → c) → NatF b → c
f |+| g = λx → case x of
  ZeroF → f
  SuccF r → g r

fact :: Nat → Nat
fact = para (succ′ zero′ |+| (mult ∘ (id >< succ′)))
```

The *apomorphism* is the primitive corecursion scheme. It is a generalization of the anamorphism and the dual of the paramorphism.

```
apo :: Functor f ⇒ (a → f (Either a (Fix f))) → a → Fix f
apo f = In ∘ fmap (apo f ||| id) ∘ f
  -- Alternative
apo :: Functor f ⇒ (a → f (Either a (Fix f))) → a → Fix f
apo f = ana (f ||| (fmap Right ∘ out)) ∘ Left

append :: List a → List a → List a
append l = apo (((fmap Right (out l)) |−| ConsF) ∘ fmap Left ∘ out)
```

3. (3) For this question, refer to the paper "Data Types à la Carte" and/or the presentation by Wout Elsinghorst.

   We wish to distinguish input and output `IO` operations by type. For this exercise, we want to define everything necessary for the following two functions:

```
getLine :: (Input :<: f) ⇒ Term f String
getLine = inject (GetLine Pure)

putStrLn :: (Output :<: f) ⇒ String → Term f ()
putStrLn s = inject (PutStrLn s (Pure ()))
```

Then, we can write `IO` functions that use only `getLine` or only `putStrLn` or a combination of both, as in the following example:

```
prompt :: Term (Input :+: Output) ()
prompt = do
  s <- getLine
  putStrLn ("You wrote: " ++ s)
test_prompt :: IO ()
test_prompt = exec prompt
```

Define the datatypes `Input` and `Output` and a minimal "à la carte" library that implements the necessary functionality. Be sure that all code is included and that the above code works.

Solution. The datatypes follow:

```
data Input  r = GetLine (String -> r)
data Output r = PutStrLn String r
```

They require the following instances:

```
instance Functor Input where
  fmap f (GetLine g) = GetLine (f . g)
instance Functor Output where
  fmap f (PutStrLn s m) = PutStrLn s (f m)
instance Exec Input where
  execAlgebra (GetLine f) = Prelude.getLine >>= f
instance Exec Output where
  execAlgebra (PutStrLn s io) = Prelude.putStrLn s >> io
```

The rest of the library is mostly from the paper:

```haskell
data (f :+: g) e = Inl (f e) | Inr (g e)

instance (Functor f, Functor g) => Functor (f :+: g) where
  fmap f (Inl x) = Inl (fmap f x)
  fmap f (Inr x) = Inr (fmap f x)

class (Functor sub, Functor sup) => sub :<: sup where
  inj :: sub a -> sup a
instance Functor f => f :<: f where
  inj = id
instance (Functor f, Functor g) => f :<: (f :+: g) where
  inj = Inl
instance (Functor f, Functor g, Functor h, f :<: g) => f :<: (h :+: g) where
  inj = Inr . inj

data Term f a = Pure a | Impure (f (Term f a))

instance Functor f => Functor (Term f) where
  fmap f (Pure x)   = Pure (f x)
  fmap f (Impure t) = Impure (fmap (fmap f) t)

instance Functor f => Monad (Term f) where
  return = Pure
  Pure x   >>= f = f x
  Impure t >>= f = Impure (fmap (>>=f) t)

inject :: (g :<: f) => g (Term f a) -> Term f a
inject = Impure . inj

foldTerm :: Functor f => (a -> b) -> (f b -> b) -> Term f a -> b
foldTerm pure imp (Pure x)   = pure x
foldTerm pure imp (Impure t) = imp (fmap (foldTerm pure imp) t)

class Functor f => Exec f where
  execAlgebra :: f (IO a) -> IO a

  -- This instance is omitted from the paper.
instance (Exec f, Exec g) => Exec (f :+: g) where
  execAlgebra (Inl x) = execAlgebra x
  execAlgebra (Inr x) = execAlgebra x

exec :: Exec f => Term f a -> IO a
exec = foldTerm return execAlgebra
```