Dept. of Information and Computing Sciences, Utrecht University

Theory of Programming and Types 2015

Exercise Set 3

Johan Jeuring

June 2015

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 via submit on or before the following deadline:

   17 June, 2015

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 (GList f (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
  Succ   :: Exp               → Exp
  If     :: Exp → Exp → Exp → Exp
```

   What is the kind of each datatype?

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.
   - `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.

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

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

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
```

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
```

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

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
```

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'`.

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

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.

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 (or higher, I didn't check this), available here: `http://www.cse.chalmers.se/~nad/software/lib-0.7.tar.gz`. Implement a module Regular2Multirec.