

Generic Programming 2011

Exercise Set 4

Sean Leather

Saturday, 5 November, 2011

1 General Information

1.1 Instructions

1. Read these instructions and the notes below.
2. Read through all of the exercises below before starting, so that you have an overall idea of what is expected and how much time to plan for each.
3. Create two files called `<First><Last>4.lhs` and `<First><Last>4.lagda` with `<First>` replaced by your first name (e.g. Bertrand) and `<Last>` replaced by your surname (e.g. Russell).
4. Write a solution to each exercise in the appropriate file. Number the solutions to match the section and exercise numbers.
5. Submit your file as an email attachment to `leather@cs.uu.nl` **before 17:00 on Friday, 11 November, 2011**. You receive no credit if it is late.

1.2 Notes

- You will need to install the latest libraries from Hackage.
- 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 version 7.0.* for Haskell. This is what you get with the latest Haskell Platform. There is a newer version of GHC, 7.2.1, but be aware that you may encounter issues if you use a version different from your grader or lecturer.
- Use Agda version 2.2.10 for Agda.
- All code should type-check when loading the file into GHCi or the Agda interpreter (or Agda mode in Emacs).
- The maximum possible score is 10. Each exercise has a number in parentheses representing its maximum possible score.

Good luck!

2 Exercises

2.1 Written

1. (2) For this exercise, refer to the paper *Calculating with Lenses: Optimising Bidirectional Transformations* by Hugo Pacheco and Alcina Cunha.

Given a functor F , with $in_F :: \text{Lens } (F (\mu F)) (\mu F)$ and $out_F :: \text{Lens } (\mu F) (F (\mu F))$, and an F -algebra $\phi :: \text{Lens } (F a) a$, we have a unique function $fold_F \phi :: \text{Lens } (\mu F) a$.

The uniqueness law for the $fold_F$ is (where map_F is the F -map for lenses):

$$\delta = fold_F \phi \Leftrightarrow \delta \circ in_F = \phi \circ map_F \delta$$

Using this, we can prove the $fold_F$ fusion law:

$$\delta \circ \tau = \phi \circ map_F \delta \Rightarrow \delta \circ fold_F \tau = fold_F \phi$$

The proof of the $fold_F$ fusion law looks like the following in equational reasoning style. Fill in the missing part.

$$\begin{aligned} \delta \circ in_F &= \tau \circ map_F \delta \\ &\Rightarrow \{ \text{uniqueness: } \delta = fold_F \tau \} \\ fold_F \tau \circ in_F &= \tau \circ map_F (fold_F \tau) \\ &\Rightarrow \{ \text{intro } \delta \} \\ \delta \circ fold_F \tau \circ in_F &= \delta \circ \tau \circ map_F (fold_F \tau) \\ &\Rightarrow \\ \dots & \\ &\Rightarrow \\ \delta \circ fold_F \tau &= fold_F \phi \end{aligned}$$

2.2 Haskell

1. (2) For this exercise, install `Multiplate` and refer to the paper *Functor is to Lens as Applicative is to Biplate: Introducing Multiplate* by Russell O'Connor.

a) (0.8) Define an instance of `Multiplate` for the nonregular datatype `ZigZag` (a relative of the `Zig` and `Zag` we saw previously).

```
data ZigZag a b = Stop a | Cons a (ZigZag b a) deriving (Show, Typeable)
```

b) (0.6) Using `Multiplate` (and not using a nongeneric function), define the following function:

```
collect :: ZigZag a b → ([a], [b])
```

c) (0.6) Using `Multiplate`, define the following function that collects the characters in a `ZigZag` value where either or both type parameters may be `Char`:

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

2. (2) For this exercise, install `compdata` and refer to the paper *Compositional Data Types* by Patrick Bahr and Tom Hvitved.

a) (0.2) Give `Functor` and `Foldable` instances of `Nat`.

```
data Nat e = Zero | Succ e
```

Define the function `evalNat` using the function `query` or another function from the library defined with `query`.

```
evalNat :: Term Nat → Int
```

b) (0.2) Give the `Functor` instance of `Add`.

```
data Add e = Add e e
```

Define smart constructors for `Nat` (`z` and `s`) and `Add` (`.+. .`) such that they will work with `AddNat` and any other expansions of the type.

```
type AddNat = Nat :+: Add
```

```
infixl 6 . + .
```

- c) (0.4) Define an algebra that will reduce a term with addition to one with only numbers such that if $n \equiv x + y$, then the term for n replaces the term for $x + y$.

```
class RedAdd f where
  redAddAlg :: Alg f (Term Nat)
```

The algebra should result in this catamorphism:

```
redAdd :: Term AddNat → Term Nat
redAdd = cata redAddAlg
```

- d) (0.6) Give the `Functor` instance of `Mul`.

```
data Mul e = Mul e e
```

Define a smart constructor `.*` for `Mul`, such that it will work with `MulNat` and any other expansions of the type.

```
type MulNat = Mul :+: AddNat
infixl 7 .*
```

Consider the function `distMul1` that may distribute one level of multiplication over addition, assuming the two arguments come from a multiplication.

```
distMul1 :: (Add <: f, Mul <: f) ⇒ Cxt h f a → Cxt h f a → Maybe (Cxt h f a)
distMul1 a b = do Add c d ← match b
               return (a .* c .+ a .* d)
match :: (g <: f) ⇒ Cxt h f a → Maybe (g (Cxt h f a))
match (Term t) = proj t
match (Hole a) = Nothing
```

Define the algebra `distMulAlg` such that the catamorphism `distMul` distributes all multiplications over addition, reducing a term to a sum of products.

```
class DistMul f where
  distMulAlg :: Alg f (Term MulNat)
  distMul :: Term MulNat → Term MulNat
  distMul = cata distMulAlg
```

- e) (0.6) Define the following algebra that reduces a term with multiplication to a term with addition such that $x * y$ becomes x added to itself y times.

```
class ReduceMul f where
  redMulAlg :: Alg f (Term AddNat)
  redMul :: Term MulNat → Term AddNat
  redMul = cata redMulAlg
```

Using the functions above, define `evalMul` to reduce a term with multiplication and addition to an integer.

```
evalMul :: Term MulNat → Int
```

2.3 Agda

For these exercises, you will need to use Agda. The easiest way to install it is via `caball install Agda`. You will also need to download the files below from the wiki. Other files are not necessary.

- `Prelude.agda`
- `Orn.agda`
- `OrnNat.agda`
- `IxFun.agda`
- `IxFunNat.agda`

1. (2) For this exercise, refer to the paper *Modularising Inductive Families* by Hsiang–Shang Ko and Jeremy Gibbons.

- (0.5) Using the approach described in the paper and exemplified by `Nat` in the provided file, define the code `MaybeD` and the type constructor `Maybe` from the interpretation of `MaybeD`.
- (0.4) Using the type `Maybe` from the previous section, define the following functions.

$$\begin{aligned} \text{nothing}' & : \{A : \text{Set}\} \rightarrow \text{Maybe}' A \\ \text{just}' & : \{A : \text{Set}\} \rightarrow A \rightarrow \text{Maybe}' A \\ \text{maybe}' & : \{A B : \text{Set}\} \rightarrow B \rightarrow (A \rightarrow B) \rightarrow \text{Maybe}' A \rightarrow B \\ _<^{**}>_ & : \{A B : \text{Set}\} \rightarrow \text{Maybe}' A \rightarrow \text{Maybe}' (A \rightarrow B) \rightarrow \text{Maybe}' B \end{aligned}$$

- (0.5) Define `EitherO` as an ornamented description of `MaybeD` and define the type constructor `Either` from the interpretation of `EitherO`.
- (0.4) Using the type `Either` from the previous section, define the following functions.

$$\begin{aligned} \text{left}' & : \{A B : \text{Set}\} \rightarrow A \rightarrow \text{Either}' A B \\ \text{right}' & : \{A B : \text{Set}\} \rightarrow B \rightarrow \text{Either}' A B \\ \text{either}' & : \{A B C : \text{Set}\} \rightarrow (A \rightarrow C) \rightarrow (B \rightarrow C) \rightarrow \text{Either}' A B \rightarrow C \end{aligned}$$

- (0.2) Use the library to define the following function.

$$\text{EitherToMaybe} : \{A : \text{Set}\} \{B : \text{Set}\} \rightarrow \text{Either}' A B \rightarrow \text{Maybe}' B$$

2. (2) For this exercise, refer to the paper *Generic Programming with Indexed Functors* by Andres Löh and José Pedro Magalhães.

- a) (1) Using the provided files, define the indexed functor code 'Maybe' for `Maybe` as well as the functions `fromMaybe` and `toMaybe` which translate terms between the `Maybe` datatype and the interpreted code.
- b) (1) Give the isomorphism between the datatype `Maybe` and the interpreted 'Maybe'. The isomorphism is defined by instantiating a record of the `__isom__` type.