

## INFOAFP Assignments

# AFP Assignment 4

Deadline: Jan 11, 2013

### General remarks

- Mail your solution to `doaitse@swierstra.net`, with in the subject "Assign-2-4: *name1* and *name2*" and containing a zip file with name `2012-2-;name1;-;name2;`.
- Team size: preferably 2, but 1 is possible.
- For programs: Programs that are not type correct may not be graded. Programming style influences the grade.
- For text: Submit plain text or PDF, *not* HTML or Word.
- Gathering information on the internet is okay, but copying entire solutions from the internet (or elsewhere) is not allowed.
- You can make a Cabal package again, or just submit a zip file.

1 (10%). Here is a nested datatype for square matrices:

```
type Square a = Square' Nil a
data Square' t a = Zero (t (t a)) | Succ (Square' (Cons t) a)
data Nil a = Nil
data Cons t a = Cons a (t a)
```

Give Haskell code that represents the following two square matrices as elements of the *Square* datatype:

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \text{ and } \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

Note: you don't have to define any functions for the *Square* datatype. Defining sensible functions for *Square* (even *show*) is not entirely trivial and might be the topic of a later assignment.

2 (10%). Write a function

$$\text{forceBoolList} :: [\text{Bool}] \rightarrow r \rightarrow r$$

that completely forces a list of Boolean values *without using seq*. Note that pattern matching drives evaluation.

Explain why the function *forceBoolList* has the type as specified above and not:

$$\text{forceBoolList} :: [\text{Bool}] \rightarrow [\text{Bool}]$$

and why *seq* is defined as it is, and

$$\begin{aligned} \text{force} &:: a \rightarrow a \\ \text{force } a &= \text{seq } a \ a \end{aligned}$$

is useless.

3 (10%). Define a function *count* such that the following program is well-typed

$$\begin{aligned} \text{test} &:: [\text{Int}] \\ \text{test} &= [\text{count}, \text{count } 1 \ 2 \ 3, \text{count } "" \ [\text{True}, \text{False}] \ \text{id } (+)] \end{aligned}$$

and evaluates to  $[0, 0, 0]$ . In other words, *count* should accept an arbitrary number of arguments (of arbitrary types), and just always return 0.

Then redefine the function *count* such that *test* evaluates to  $[0, 3, 4]$ , i.e., *count* should return the number of arguments.

Please submit both versions of the function.

Hint: no language extensions are required to solve this exercise.

4 (10%). A curious fact is that Haskell's type system (even that of Haskell without extensions) has exponential space and time complexity. However, the worst case rarely occurs in practice such that the run-time behaviour of the type checker generally is acceptable. Define a family of Haskell expressions such that the type (i.e., the size of the type expression) grows exponentially in the size of the program. Note that if the type is highly repetitive, the type can internally be represented using sharing. However, different type variables cannot be shared. So, to get a truly large type, you have to try to get as many different type variables as possible. If you find your solution on the internet explain how it works!

5 (20%). Recall the datatype of square matrices:

```
type Square      = Square' Nil
data Square' t a = Zero (t (t a)) | Succ (Square' (Cons t) a)
data Nil a      = Nil
data Cons t a   = Cons a (t a)
```

Note that we have eta-reduced the definition of *Square*. This turns out to be necessary in the end where we will mention it again.

Let's investigate how we can derive an equality function on square matrices. We do so very systematically by deriving an equality function for each of the four types. We follow a simple, yet powerful principle: type abstraction corresponds to term abstraction, and type application corresponds to term application.

What does this mean? If a type  $f$  is parameterized over an argument  $a$ , then in general, we have to know how equality is defined on  $a$  in order to define equality on  $f a$ . Therefore we define

$$\begin{aligned} eqNil &:: (a \rightarrow a \rightarrow Bool) \rightarrow (Nil a \rightarrow Nil a \rightarrow Bool) \\ eqNil eqA Nil Nil &= True \end{aligned}$$

In this case, the  $a$  is not used in the definition of  $Nil$ , so it is not surprising that we do not use  $eqA$  in the definition of  $eqNil$ . But what about  $Cons$ ? The datatype  $Cons$  has two arguments  $t$  and  $a$ , so we expect two arguments to be passed to  $eqCons$ , something like

$$eqCons eqT eqA (Cons x xs) (Cons y ys) = eqA x y \wedge \dots$$

But what should the type of  $eqT$  be? The  $t$  is of kind  $* \rightarrow *$ , so it can't be  $t \rightarrow t \rightarrow Bool$ . We can argue that we should use  $t a \rightarrow t a \rightarrow Bool$ , because we use  $t$  applied to  $a$  in the definition of  $Cons$ . However, a better solution is to recognise that, being a type constructor of kind  $* \rightarrow *$ , an equality function on  $t$  should take an equality function on its argument as a parameter. And, moreover, it does not matter what this parameter is! A function like  $eqNil$  is polymorphic in type  $a$ , so let us require that  $eqT$  is polymorphic in the argument type as well:

$$\begin{aligned} eqCons &:: (\forall b.(b \rightarrow b \rightarrow Bool) \rightarrow (t b \rightarrow t b \rightarrow Bool)) \rightarrow \\ &\quad (a \rightarrow a \rightarrow Bool) \rightarrow \\ &\quad (Cons t a \rightarrow Cons t a \rightarrow Bool) \\ eqCons eqT eqA (Cons x xs) (Cons y ys) &= eqA x y \wedge eqT eqA xs ys \end{aligned}$$

Now you can see how we apply  $eqT$  to  $eqA$  when we want equality at type  $t a$  – the type application corresponds to term application.

*Task.* A type with a  $\forall$  on the inside requires the extension `RankNTypes` to be enabled. Try to understand what the difference is between a function of the type of  $eqCons$  and a function with the same type but the  $\forall$  omitted. Can you omit the  $\forall$  in the case of  $eqCons$  and does the function still work?

Now, on to  $Square'$ . The type of  $eqSquare'$  follows exactly the same idea as the type of  $eqCons$ :

$$\begin{aligned} eqSquare' &:: (\forall b.(b \rightarrow b \rightarrow Bool) \rightarrow (t b \rightarrow t b \rightarrow Bool)) \rightarrow \\ &\quad (a \rightarrow a \rightarrow Bool) \rightarrow \\ &\quad (Square' t a \rightarrow Square' t a \rightarrow Bool) \end{aligned}$$

We now for the first time have more than one constructor, so we actually have to give multiple cases. Let us first consider comparing two applications of  $Zero$ :

$$eqSquare' eqT eqA (Zero xs) (Zero ys) = eqT (eqT eqA) xs ys$$

Note how again the structure of the definition follows the structure of the type. We have a value of type  $t$  ( $t a$ ). We compare it using  $eqT$ , passing it an equality function for values of type  $t a$ . How? By using  $eqT eqA$ .

The remaining cases are as follows:

$$\begin{aligned} eqSquare' eqT eqA (Succ xs) (Succ ys) &= eqSquare' (eqCons eqT) eqA xs ys \\ eqSquare' eqT eqA _ _ &= False \end{aligned}$$

The idea is the same – let the structure of the recursive calls follow the structure of the type.

*Task.* Again, try removing the  $\forall$  from the type of  $eqSquare'$ . Does the function still typecheck? Try to explain!

Now we're done:

$$\begin{aligned} eqSquare &:: (a \rightarrow a \rightarrow Bool) \rightarrow Square a \rightarrow Square a \rightarrow Bool \\ eqSquare &= eqSquare' eqNil \end{aligned}$$

Test the function. We can now also give an *Eq* instance for *Square* – this requires the minor language extension *TypeSynonymInstances*, because for some stupid reason, Haskell 98 does not allow type synonyms like *Square* to be used in instance declarations:

```
instance Eq a => Eq (Square a) where
    (≡) = eqSquare (≡)
```

*Task.* Systematically follow the scheme just presented in order to define a *Functor* instance for square matrices. I.e., derive a function *mapSquare* such that you can define

```
instance Functor Square where
    fmap = mapSquare
```

This instance requires *Square* to be defined in eta-reduced form in the beginning, because Haskell does not allow partially applied type synonyms.

6 (20%). Consider the following datatype:

```
data GP a = End a
         | Get (Int → GP a)
         | Put Int (GP a)
```

A value of type *GP* can be used to describe programs that read and write integer values and return a final result of type  $a$ . Such a program can end immediately (*End*). If it reads an integer, the rest of the program is described as a function depending on this integer (*Get*). If the program writes an integer (*Put*), the value of that integer and the rest of the program are recorded.

The following expression describes a program that continuously reads integers and prints them:

$$echo = Get (\lambda n \rightarrow Put n echo)$$

*Task.* Write a function

$$run :: GP a \rightarrow IO a$$

that can run a *GP*-program in the *IO* monad. A *Get* should read an integer from the console, and *Put* should write an integer to the console.

Here is an example run from GHCi:

```
>>> run echo
? 42
42
? 28
28
? 1
1
? - 5
- 5
? Interrupted.
>>>
```

[To better distinguish inputs from outputs, this version of *run* prints a question mark when expecting an input.]

*Task.* Write a *GP*-program *add* that reads two integers, writes the sum of the two integers, and ultimately returns ().

*Task.* Write a *GP*-program *accum* that reads an integer. If the integer is 0, it returns the current total. If the integer is not 0, it adds the integer to the current total, prints the current total, and starts from the beginning.

*Task.* Instead of running a *GP*-program in the *IO* monad, we can also simulate the behaviour of such a program by providing a (possibly infinite) list of input values. Write a function

$$simulate :: GP a \rightarrow [Int] \rightarrow (a, [Int])$$

that takes such a list of input values and returns the final result plus the (possibly infinite) list of all the output values generated.

A map function for *GP* can be defined as follows:

```
instance Functor GP where
  fmap f (End x)   = End (f x)
  fmap f (Get g)   = Get (fmap f \circ g)
  fmap f (Put n x) = Put n (fmap f x)
```

*Task.* Define sensible instances of *Monad* and *MonadState* for *GP*. How is the behaviour of the *MonadState* instance for *GP* different from the usual *State* type?

7 (20%). Consider the following module:

```
import Control.Monad.Reader
import System.Random
one :: Int
one = 1
two :: Int
two = 2
randomN :: (RandomGen g) => Int -> g -> Int
randomN n g = (fst (next g) `mod` (two * n + one)) - n
sizedInt = do
    n ← ask
    g ← lift ask
    return (randomN n g)
```

What is the most general type of *sizedInt*? (Note that type inference may not work for *sizedInt*, but you should be able to explain the error message by now and know how to fix it. Note further that the most general type will only be accepted by GHC in a type signature when `FlexibleContexts` are enabled.)

Assuming this most general type, perform an evidence translation for all the overloading involved in the functions *randomN* and *sizedInt*. First, define the record types for the classes involved. You can *ignore* the fact that literals and arithmetic operations are overloaded and just use *one* and *two* as monomorphic integers. You only have to include those methods in the records that are actually used in the program above. However, you *should* consider the desugaring (you may simplify and ignore the **let** statements for the patterns) of the **do** notation to the monad operations, as well as the overloaded *ask* and *lift* functions. In order to define the record types correctly, you must enable the `PolymorphicComponents` or `RankNTypes` language extensions to allow polymorphic fields in datatypes.

Then translate *randomN* and *sizedInt* similar to the translation on the slides. You are allowed to introduce local abbreviations using **let** and **where** for often-used expressions. The resulting program must, of course, still be type correct in Haskell.