INFOAFP Assignments

AFP Assignment 2

Deadline: Thursday, 19 February 2009, 20:00

General remarks

- Submission via the "Submit system".
- Team size: 1 or 2.
- 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 is not allowed.

1 (30%)**.** Consider the following definitions:

$$
\begin{aligned}
&\textbf{data}\ \mathit{Tree}\ a = \mathit{Leaf}\ a \\
&\qquad\qquad \mid\ \mathit{Node}\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a) \\
&\quad \textbf{deriving}\ \mathit{Show} \\
&\mathit{size} :: \mathit{Tree}\ a \to \mathit{Int} \\
&\mathit{size}\ (\mathit{Leaf}\ a) \quad = 1 \\
&\mathit{size}\ (\mathit{Node}\ l\ r) = \mathit{size}\ l + \mathit{size}\ r \\
&\mathit{length} :: [a] \to \mathit{Int} \\
&\mathit{length}\ [\,] \qquad = 0 \\
&\mathit{length}\ (x : xs) = 1 + \mathit{length}\ xs \\
&\mathit{flatten} :: \mathit{Tree}\ a \to [a] \\
&\mathit{flatten}\ (\mathit{Leaf}\ a) \quad = [a] \\
&\mathit{flatten}\ (\mathit{Node}\ l\ r) \ = \mathit{flatten}\ l \mathbin{+\!\!+} \mathit{flatten}\ r \\
&(\mathbin{+\!\!+}) :: [a] \to [a] \to [a] \\
&[\,] \qquad \mathbin{+\!\!+} ys \quad = ys \\
&(x : xs) \mathbin{+\!\!+} ys \quad = x : (xs \mathbin{+\!\!+} ys)
\end{aligned}
$$

Prove the following Theorem using equational reasoning:

$$\forall (t :: \mathit{Tree}\ a).\mathit{length}\ (\mathit{flatten}\ t) \equiv \mathit{size}\ t$$

Note that using the induction principle on trees, it is sufficient to show the following two cases:

$$\forall (x :: a).\mathit{length}\ (\mathit{flatten}\ (\mathit{Leaf}\ x)) \equiv \mathit{size}\ (\mathit{Leaf}\ x)$$

and

$$\begin{array}{l}\forall (l :: \mathit{Tree}\ a)\ (r :: \mathit{Tree}\ a). \\ \quad (\ \mathit{length}\ (\mathit{flatten}\ l) \equiv \mathit{size}\ l \\ \quad \wedge\ \mathit{length}\ (\mathit{flatten}\ r) \equiv \mathit{size}\ r \\ \quad)\ \ \rightarrow \mathit{length}\ (\mathit{flatten}\ (\mathit{Node}\ l\ r)) \equiv \mathit{size}\ (\mathit{Node}\ l\ r)\end{array}$$

To prove the Theorem, you will need to prove the following Lemma first:

$$\forall (xs :: [a])\ (ys :: [a]).\mathit{length}\ (xs +\!\!+\ ys) = \mathit{length}\ xs + \mathit{length}\ ys$$

You may use facts about $(+)$ such that $(+)$ is associative or that 0 is the neutral element of addition.

2 (40%)**.** Write a Haskell module containing all the declarations from the previous exercise (you will have to hide some definitions from the Prelude to be prevent ambiguities).

Also import QuickCheck.

Define

$$\textbf{instance}\ \mathit{Arbitrary}\ a \Rightarrow \mathit{Arbitrary}\ (\mathit{Tree}\ a)$$

suitably so that you can test properties parameterized by trees.

Define QuickCheck properties for both the Theorem and the Lemma from the previous exercise.

Define a function *revTree* that reverses a tree, and define QuickCheck properties that state that

- reversing a tree and flattening it is the same as flattening a tree and reversing the resulting list,
- reversing a tree twice results in the original tree,
- the size of a reversed tree is the same as the size of the original tree.

Verify that your *revTree* function passes all three properties.

3 (30%)**.** Recursion can be encoded in the Lambda Calculus using so-called *fixed point* combinators: here is one

$$\mathit{fix} = \lambda f \rightarrow (\lambda x \rightarrow f\ (x\ x))\ (\lambda x \rightarrow f\ (x\ x))$$

Note that *fix* is not recursive itself. Nevertheless, it can be used to define a recursive function. The key property of *fix* is that

$$\mathit{fix}\ f \equiv f\ (\mathit{fix}\ f)$$

We can show this property using equational reasoning:

$$\begin{array}{ll}
 & \mathit{fix}\ f \\
\equiv & \quad \{ \text{ Definition of } \mathit{fix} \ \} \\
 & (\lambda f \to (\lambda x \to f\ (x\ x))\ (\lambda x \to f\ (x\ x)))\ f \\
\equiv & \quad \{ \text{ Reducing the outermost application } \} \\
 & (\lambda x \to f\ (x\ x))\ (\lambda x \to f\ (x\ x)) \\
\equiv & \quad \{ \text{ Reducing the outermost application } \} \\
 & f\ ((\lambda x \to f\ (x\ x))\ (\lambda x \to f\ (x\ x))) \\
\equiv & \quad \{ \text{ Abstracting from } f \text{ in the argument to } f \ \} \\
 & f\ ((\lambda f \to (\lambda x \to f\ (x\ x))\ (\lambda x \to f\ (x\ x)))\ f) \\
\equiv & \quad \{ \text{ Definition of } \mathit{fix} \ \} \\
 & f\ (\mathit{fix}\ f)
\end{array}$$

Note how we applied a call-by-name reduction strategy by always selecting the outermost redex.

Task. Curiously, the definition of *fix* does not type-check in Haskell. Put the definition of *fix* into a Haskell module, run GHCi and try to explain the type error message.

Task. Nevertheless, we can define *fix* in Haskell using Haskell's recursion, by defining it via its key property:

$$\mathit{fix}\ f = f\ (\mathit{fix}\ f)$$

What is the type of *fix*?

Task. Rewrite the length function

$$\begin{array}{l}
\mathit{len} :: [a] \to \mathit{Int} \\
\mathit{len}\ \mathit{xs} = \textbf{if}\ \mathit{null}\ \mathit{xs}\ \textbf{then}\ 0 \\
\qquad\qquad\qquad\qquad \textbf{else}\ 1 + \mathit{len}\ (\mathit{tail}\ \mathit{xs})
\end{array}$$

such that the definition makes use of *fix* and is not recursive anymore.

4 (bonus)**.** We can define the fixed point combinator from the previous exercise in Haskell if we help the type checker along. This requires introducing the following (recursive!) datatype:

$$\textbf{data}\ F\ a = F\ \{\mathit{unF} :: F\ a \to a\}$$

Try to define *fix* now, without using recursion – just take the definition as a lambda term and insert applications of *F* and *unF* as needed.