Prof. Dr. Ralf Hinze
M.Sc. Sebastian Schweizer

TU Kaiserslautern
Fachbereich Informatik
AG Programmiersprachen

Functional Programming: Exercise 4

Sheet published: Thursday, May 9th
Submission deadline: Wednesday, May 15th, 3:00 pm

Exercise 4.1 (Skeleton: `BinaryTrees.hs`). Consider the following data type:

$$\textbf{data}\ \mathit{Tree}\ \mathit{elem} = \mathit{Empty} \mid \mathit{Node}\ (\mathit{Tree}\ \mathit{elem})\ \mathit{elem}\ (\mathit{Tree}\ \mathit{elem})$$

a) Write a function $\mathit{left} :: \mathit{Tree}\ a \to \mathit{Maybe}\ a$ that returns the left-most element in the given tree, or *Nothing* if the tree is empty.

b) Write a function $\mathit{reverseTree} :: \mathit{Tree}\ a \to \mathit{Tree}\ a$ that returns a tree whose inorder traversal is the reverse of the original trees's inorder traversal.
The specification is: $\mathit{inorder}\ (\mathit{reverseTree}\ t) == \mathit{reverse}\ (\mathit{inorder}\ t)$.
Hint: Solve it in a graphical way first, then write the code.

Exercise 4.2 (Skeleton: `Calculus.hs`). Lisa Lista's younger brother just went through differential calculus at high school. She decides to implement derivatives in Haskell to be able to easily double-check his homework solutions. To this end she introduces the following datatype:

$$\begin{array}{l}
\textbf{infixl}\ 6 :+: \\
\textbf{infixl}\ 7 :*: \\
\textbf{infixl}\ 8 :\hat{}: \\
\textbf{infixl}\ 9 :\circ:
\end{array}$$

$$\begin{array}{lll}
\multicolumn{2}{l}{\textbf{data}\ \mathit{Function}} & \\
\quad = & \mathit{Const}\ \mathit{Rational} & \text{— constant function} \\
\quad \mid & \mathit{Id} & \text{— identity} \\
\quad \mid & \mathit{Function} :+: \mathit{Function} & \text{— addition of functions} \\
\quad \mid & \mathit{Function} :*: \mathit{Function} & \text{— multiplication of functions} \\
\quad \mid & \mathit{Function} :\hat{}: \mathit{Integer} & \text{— power with constant exponent} \\
\quad \mid & \mathit{Function} :\circ: \mathit{Function} & \text{— composition of functions} \\
\multicolumn{2}{l}{\quad \textbf{deriving}\ (\mathit{Show})} &
\end{array}$$

The idea is that each element of *Function* *represents* a function over the rationals[1] e.g. Id represents $\backslash x \to x$, $\mathit{Const}\ r$ represents $\backslash x \to r$, $\mathit{Id} :\hat{}: (-2)$ represents $\backslash x \to x\hat{}\hat{}(-2)$, and $(\mathit{Id} :\hat{}: 5) :\circ: (\mathit{Id} :+: \mathit{Const}\ 3)$ represents $\backslash x \to (x+3)\hat{}\hat{}5$ (which could also be written as $(\backslash x \to x\hat{}\hat{}5) \circ (\backslash x \to x+3)$). Note that we use $(\hat{}\hat{})$ instead of $(\hat{})$ because the former allows for negative exponents, where $x\hat{}\hat{}(-n) == 1\,/\,(x\hat{}\hat{}n)$.

In general, if $e1$ represents $f1$ and $e2$ represents $f2$, then $e1 :+: e2$ represents the function $\backslash x \to (f1\ x) + (f2\ x)$, $e1 :*: e2$ represents the function $\backslash x \to (f1\ x) * (f2\ x)$, $e1 :\circ: e2$ represents the function $\backslash x \to f1\ (f2\ x)$, and $e1 :\hat{}: n$ represents the function $\backslash x \to (f1\ x)\hat{}\hat{}n$.

a) Define a function $\mathit{apply} :: \mathit{Function} \to (\mathit{Rational} \to \mathit{Rational})$ that applies the representation of a function to a given value. In a sense, *apply* maps syntax to semantics: the representation of a function is mapped to the actual Haskell function.

[1] The type *Rational* ins Haskell represents rational numbers exactly as the ratio of two *Integers*.

b) Define a function $\mathit{derive} :: \mathit{Function} \to \mathit{Function}$ that computes the derivative of a function. Hint: For the derivation of $:\hat{}:$ you probably need to convert an $\mathit{Integer}$ value to $\mathit{Rational}$, you can use the function $\mathit{toRational}$ there.

c) After Lisa has captured the rules of derivatives as a Haskell function, she tests the implementation on a few simple examples. The initial results are not too encouraging:

$$\ggg\ \mathit{derive}\ (\mathit{Const}\ 1 :+: \mathit{Const}\ 2 :*: \mathit{Id})$$
$$\mathit{Const}\ (0\,\%\,1) :+: (\mathit{Const}\ (0\,\%\,1) :*: \mathit{Id} :+: \mathit{Const}\ (2\,\%\,1) :*: \mathit{Const}\ (1\,\%\,1))$$
$$\ggg\ \mathit{derive}\ (\mathit{Id} :\circ: \mathit{Id} :\circ: \mathit{Id})$$
$$(\mathit{Const}\ (1\,\%\,1) :\circ: \mathit{Id} :*: \mathit{Const}\ (1\,\%\,1)) :\circ: \mathit{Id} :*: \mathit{Const}\ (1\,\%\,1)$$

Implement a function $\mathit{simplify} :: \mathit{Function} \to \mathit{Function}$ that simplifies the representation of a function using the laws of algebra. (This is a lot harder than it sounds!)
Hint: use smart constructors.

Exercise 4.3 (Skeleton: `Fold.hs`)**.** Use the order functions foldl and foldr to define:

a) a function $\mathit{allTrue} :: [\mathit{Bool}] \to \mathit{Bool}$ that determines whether every element of a list of Booleans is true;

b) a function $\mathit{allFalse}$ that similarly determines whether every element of a list of Booleans is false;

c) a function $\mathit{member} :: (\mathit{Eq}\ a) \Rightarrow a \to [a] \to \mathit{Bool}$ that determines whether a specified element is contained in a given list;

d) a function $\mathit{smallest} :: [\mathit{Int}] \to \mathit{Int}$ that calculates the smallest value in a list of integers;

e) a function $\mathit{largest} :: [\mathit{Int}] \to \mathit{Int}$ that similarly calculates the largest value in a list of integers.

If both recursion schemes are applicable, which one is preferable in terms of running time?