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

TU Kaiserslautern
Fachbereich Informatik
AG Programmiersprachen

Functional Programming: Exercise 7

Sheet published: Wednesday, June 5th
Submission deadline: Wednesday, June 12th, 12:00 noon

Exams The exam dates are July 15th – 18th. Please write an email to our secretary (stengel@cs.uni-kl.de) for an appointment and don't forget to register the exam at your examination office at least two weeks in advance.

Submission The first two tasks are proofs, so you can submit a scan or photo of your handwritten solution, a nice solution made in LaTeX, or some plaintext file.

Exercise 7.1 (Induction). Consider the following program for binary search trees:

```
data Tree a = Leaf | Node (Tree a) a (Tree a)
insert :: Ord a => a -> Tree a -> Tree a
insert x Leaf  = Node Leaf x Leaf
insert x (Node l e r)
  | x <= e     = Node (insert x l) e r
  | otherwise  = Node l e (insert x r)
```

A binary tree is a search tree if and only if for each $Node\ l\ e\ r$, all elements in l are $\leqslant e$ and all elements in r are $>e$:

```
isSearchTree :: Ord a => Tree a -> Bool
isSearchTree Leaf          = True
isSearchTree (Node l e r)  = allT (\x -> x <= e) l
                          && allT (\x -> x > e) r
                          && isSearchTree l
                          && isSearchTree r
allT :: (a -> Bool) -> Tree a -> Bool
allT _ Leaf          = True
allT f (Node l e r) = f e && allT f l && allT f r
```

Using a structural induction on the $Tree$ data type, show that

a) $\forall t, y, f \quad f\ y\ \&\&\ allT\ f\ t \implies allT\ f\ (insert\ y\ t)$

b) $\forall t, y \qquad isSearchTree\ t \implies isSearchTree\ (insert\ y\ t)$

Annotate each step of your proof with a justification (for example "def. insert"). You can write $isST$ instead of $isSearchTree$.

Exercise 7.2 (Fusion)**.** Like *foldr* captures a common recursion scheme (canned recursion), *fusion* captures a common induction scheme (canned induction).

$$\begin{array}{llll} f\ (\mathit{foldr}\ (\lhd)\ e\ \mathit{xs}) = \mathit{foldr}\ (\blacktriangleleft)\ (f\ e)\ \mathit{xs} & \Longleftarrow & f\ (a \lhd b) = a \blacktriangleleft f\ b \\ f \circ \mathit{foldr}\ (\lhd)\ e \qquad\ \ = \mathit{foldr}\ (\blacktriangleleft)\ (f\ e) & \Longleftarrow & f \circ (a\lhd)\ = (a\ \blacktriangleleft) \circ f \end{array}$$

Many functions can be expressed in terms of *foldr*. This allows us to use fusion with those functions, too. For example:

$$\begin{array}{ll} x +\!\!+\ y & = \mathit{foldr}\ (:)\ y\ x \\ \mathit{and} & = \mathit{foldr}\ (\&\&)\ \mathit{True} \\ \mathit{concat} & = \mathit{foldr}\ (+\!\!+)\ [\,] \\ \mathit{length} & = \mathit{foldr}\ (\backslash x\ n \to 1 + n)\ 0 \\ \mathit{map}\ f & = \mathit{foldr}\ (\backslash x\ \mathit{xs} \to f\ x : \mathit{xs})\ [\,] \\ \mathit{or} & = \mathit{foldr}\ (||)\ \mathit{False} \\ \mathit{sum} & = \mathit{foldr}\ (+)\ 0 \end{array}$$

a) Using fusion, prove that $2 * (\mathit{foldr}\ (+)\ 0\ \mathit{xs}) = \mathit{foldr}\ (\backslash a\ x \to 2 * a + x)\ 0\ \mathit{xs}$.

$$\begin{array}{ll} & 2 * (\mathit{foldr}\ (+)\ 0\ \mathit{xs}) = \mathit{foldr}\ (\backslash a\ x \to 2 * a + x)\ 0\ \mathit{xs} \\ \Longleftarrow & \{\ \mathit{foldr}\ \text{fusion}\ \} \\ & 2 * 0 = 0 \wedge 2 * (a + b) = 2 * a + (2 * b) \\ \Longleftrightarrow & \{\ \text{distributive law}\ \} \\ & \mathit{True} \end{array}$$

b) Prove the *foldr*-*map* fusion law:

$$\mathit{foldr}\ (\rhd)\ e \circ \mathit{map}\ f = \mathit{foldr}\ (\backslash a\ b \to f\ a \rhd b)\ e$$

Do *not* use induction. Instead, apply fusion making use of the fact that *map* can be defined in terms of *foldr*.

$$\begin{array}{ll} & \mathit{foldr}\ (\rhd)\ e \circ \mathit{map}\ f = \mathit{foldr}\ (\backslash a\ b \to f\ a \rhd b)\ e \\ \Longleftrightarrow & \{\ \mathit{map}\ f = \mathit{foldr}\ (\backslash x\ \mathit{xs} \to f\ x : \mathit{xs})\ [\,]\ \} \\ & \mathit{foldr}\ (\rhd)\ e \circ \mathit{foldr}\ (\backslash x\ \mathit{xs} \to f\ x : \mathit{xs})\ [\,] = \mathit{foldr}\ (\backslash a\ b \to f\ a \rhd b)\ e \\ \Longleftarrow & \{\ \mathit{foldr}\ \text{fusion}\ \} \\ & \mathit{foldr}\ (\rhd)\ e\ [\,] = e \wedge (\mathit{foldr}\ (\rhd)\ e) \circ (\backslash \mathit{xs} \to f\ a : \mathit{xs}) = (\backslash b \to f\ a \rhd b) \circ (\mathit{foldr}\ (\rhd)\ e) \\ \Longleftrightarrow & \{\ \text{universal property for}\ \mathit{foldr}\text{, function composition, extensionality}\ \} \\ & \mathit{foldr}\ (\rhd)\ e\ (f\ a : \mathit{xs}) = f\ a \rhd (\mathit{foldr}\ (\rhd)\ e\ \mathit{xs}) \\ \Longleftrightarrow & \{\ \text{universal property for}\ \mathit{foldr}\ \} \\ & \mathit{True} \end{array}$$

c) Use *foldr*-*map* fusion to prove that $length = sum \circ map\ (const\ 1)$.
Hint: Substitute *length* and *sum* with the definitions given above.

$$
\begin{array}{ll}
& sum \circ map\ (const\ 1) = length \\
\iff & \{\ sum = foldr\ (+)\ 0,\ length = foldr\ (\backslash x\ n \to 1 + n)\ 0\ \} \\
& foldr\ (+)\ 0 \circ map\ (const\ 1) = foldr\ (\backslash x\ n \to 1 + n)\ 0 \\
\Longleftarrow & \{\ foldr\text{-}map \text{ fusion}\ \} \\
& 1 + n = const\ 1\ x + n \\
\iff & \{\ \text{def. } const\ \} \\
& 1 + n = 1 + n \\
\iff & \\
& True
\end{array}
$$

d) Again prove that $length\ xs = sum\ (map\ (const\ 1)\ xs)$, this time by doing a structural induction on the list data type. Use the definitions:

$$
\begin{array}{lll}
length\ [\,] & = 0 \\
length\ (x : xs) & = 1 + length\ xs \\
sum\ \ [\,] & = 0 \\
sum\ \ (x : xs) & = x + sum\ xs \\
map\ \_\ [\,] & = [\,] \\
map\ f\ (x : xs) & = f\ x : map\ f\ xs \\
const\ c\ \_ & = c
\end{array}
$$

Base case: $[\,]$

$$
\begin{array}{ll}
& sum\ (map\ (const\ 1)\ [\,]) \\
= & \{\ \text{def. } map\ \} \\
& sum\ [\,] \\
= & \{\ \text{def. } sum\ \} \\
& 0 \\
= & \{\ \text{def. } length\ \} \\
& length\ [\,]
\end{array}
$$

Induction step: $x : xs$
Induction Hypothesis: $\mathit{length}\ xs = \mathit{sum}\ (\mathit{map}\ (\mathit{const}\ 1)\ xs)$

$$
\begin{aligned}
&\mathit{sum}\ (\mathit{map}\ (\mathit{const}\ 1)\ (x : xs)) \\
=&\quad \{ \text{ def. } \mathit{map} \} \\
&\mathit{sum}\ (\mathit{const}\ 1\ x : \mathit{map}\ (\mathit{const}\ 1)\ xs) \\
=&\quad \{ \text{ def. } \mathit{sum} \} \\
&\mathit{const}\ 1\ x + \mathit{sum}\ (\mathit{map}\ (\mathit{const}\ 1)\ xs) \\
=&\quad \{ \text{ induction hypothesis } \} \\
&\mathit{const}\ 1\ x + \mathit{length}\ xs \\
=&\quad \{ \text{ def. } \mathit{const} \} \\
&1 + \mathit{length}\ xs \\
=&\quad \{ \text{ def. } \mathit{length} \} \\
&\mathit{length}\ (x : xs)
\end{aligned}
$$

Exercise 7.3 (Skeleton: `Streams.hs`). *Note:* many of the definitions below clash with definitions in the standard prelude. The skeleton file for this exercise uses a *hiding* clause to avoid these clashes. There are no test cases this time, since infinite data structures are hard to test. GitLab CI just checks that the file has no compile errors.

Haskell's list datatype comprises both finite and infinite lists. The type of streams defined below only contains infinite sequences.

```
data Stream elem = Cons {head :: elem, tail :: Stream elem}
infixr 5 .:
(.:) :: elem → Stream elem → Stream elem
a .: s = Cons a s
```

For example, the sequence of natural numbers is given by *from* 0 where *from* is defined:

```
from :: Integer → Stream Integer
from n = n .: from (n + 1)
```

a) Define functions

```
repeat :: a → Stream a
map    :: (a → b) → (Stream a → Stream b)
zip    :: (a → b → c) → (Stream a → Stream b → Stream c)
```

that lift elements, unary operations, and binary operations to streams. For example,

```
⟫⟫ repeat 1
⟨1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...⟩
⟫⟫ map (2∗) (from 0)
⟨0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, ...⟩
⟫⟫ zip (∗) (from 0) (from 1)
⟨0, 2, 6, 12, 20, 30, 42, 56, 72, 90, 110, 132, 156, 182, 210, 240, ...⟩
```

$$
\begin{aligned}
&\mathit{repeat}\ a = s\ \mathbf{where}\ s = a\ .\!:\ s\\
&\mathit{map}\ f\ s = f\ (\mathit{head}\ s)\ .\!:\ \mathit{map}\ f\ (\mathit{tail}\ s)\\
&\mathit{zip}\ g\ s\ t = g\ (\mathit{head}\ s)\ (\mathit{head}\ t)\ .\!:\ \mathit{zip}\ g\ (\mathit{tail}\ s)\ (\mathit{tail}\ t)
\end{aligned}
$$

b) Make *Stream elem* an instance of the *Num* type class. The general idea is that the methods are lifted to streams. This instance allows us to define the natural numbers and the Fibonacci numbers as follows:

$$
\begin{aligned}
&\mathit{nat}, \mathit{fib} :: \mathit{Stream}\ \mathit{Integer}\\
&\mathit{nat} = 0\ .\!:\ \mathit{nat} + 1\\
&\mathit{fib}\ \ = 0\ .\!:\ 1\ .\!:\ \mathit{fib} + \mathit{tail}\ \mathit{fib}
\end{aligned}
$$

$$
\begin{aligned}
&\mathbf{instance}\ \mathit{Num}\ \mathit{elem} \Rightarrow \mathit{Num}\ (\mathit{Stream}\ \mathit{elem})\ \mathbf{where}\\
&\quad (+) \qquad\qquad\ \ = \mathit{zip}\ (+)\\
&\quad (-) \qquad\qquad\ \ = \mathit{zip}\ (-)\\
&\quad (*) \qquad\qquad\ \ = \mathit{zip}\ (*)\\
&\quad \mathit{negate} \qquad\ \ = \mathit{map}\ \mathit{negate}\\
&\quad \mathit{abs} \qquad\qquad = \mathit{map}\ \mathit{abs}\\
&\quad \mathit{signum} \qquad = \mathit{map}\ \mathit{signum}\\
&\quad \mathit{fromInteger} = \mathit{repeat} \circ \mathit{fromInteger}
\end{aligned}
$$

c) Define a function

$$\mathit{take} :: \mathit{Integer} \to \mathit{Stream}\ \mathit{elem} \to [\mathit{elem}]$$

that allows us to inspect a finite portion of a stream: *take n s* returns the first n elements of s.

$$
\begin{aligned}
&\mathit{take}\ 0\ \_ = [\,]\\
&\mathit{take}\ n\ s\\
&\quad \mid n > 0 \qquad\ \ = \mathit{head}\ s : \mathit{take}\ (n-1)\ (\mathit{tail}\ s)\\
&\quad \mid \mathit{otherwise} = \mathit{error}\ \texttt{"take: negative argument"}
\end{aligned}
$$

d) The function diff computes the difference of a stream.

$$\mathit{diff} :: \mathit{Num}\ \mathit{elem} \Rightarrow \mathit{Stream}\ \mathit{elem} \rightarrow \mathit{Stream}\ \mathit{elem}$$
$$\mathit{diff}\ s = \mathit{tail}\ s - s$$

Here are some examples calls:

$\rangle\rangle\rangle\rangle\ \ \mathit{diff}\ \mathit{fib}$
$\langle 1, 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, ...\rangle$
$\rangle\rangle\rangle\rangle\ \ (\mathit{nat} - 2) * (\mathit{nat} + 3)$
$\langle -6, -4, 0, 6, 14, 24, 36, 50, 66, 84, 104, 126, 150, 176, 204, 234, ...\rangle$
$\rangle\rangle\rangle\rangle\ \ \mathit{diff}\ \mathit{it}$
$\langle 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, ...\rangle$
$\rangle\rangle\rangle\rangle\ \ \mathit{diff}\ \mathit{it}$
$\langle 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, ...\rangle$

The second difference of $(\mathit{nat} - 2) * (\mathit{nat} + 3)$ is a constant stream, as the original stream is a polynomial of degree 2.

Finite difference has a right-inverse: anti-difference or summation. Derive its definition from the specification $\mathit{diff}\ (\mathit{sum}\ s) = s$ additionally setting $\mathit{head}\ (\mathit{sum}\ s) = 0$. Here are some examples calls:

$\rangle\rangle\rangle\rangle\ \ \mathit{sum}\ (2 * \mathit{nat} + 1)$
$\langle 0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, ...\rangle$
$\rangle\rangle\rangle\rangle\ \ \mathit{sum}\ (3 * \mathit{nat}\ \hat{}\ 2 + 3 * \mathit{nat} + 1)$
$\langle 0, 1, 8, 27, 64, 125, 216, 343, 512, 729, 1000, 1331, 1728, 2197, 2744, 3375, ...\rangle$
$\rangle\rangle\rangle\rangle\ \ \mathit{sum}\ \mathit{fib}$
$\langle 0, 0, 1, 2, 4, 7, 12, 20, 33, 54, 88, 143, 232, 376, 609, 986, ...\rangle$

$$\mathit{sum}\ s = t\ \mathbf{where}\ t = 0 \mathrel{.:} s + t$$

Using **where** should be prefered over recursive functions, since **where** needs constant space while recursive functions require additional memory space whenever a recursive call is lazily evaluated.