Functional Programming
Exam, March 19, 2010

Prof. Dr. Jürgen Giesl
Carsten Fuhs

First name: ____________________________________________

Last name: ____________________________________________

Matr. number: _________________________________________

Course of study (please mark exactly one):
  ○ Bachelor of Informatik – Wahlpflicht
  ○ Master of Mathematik

• On every sheet please give your first name, last name, and matriculation number.
• You must solve the exam without consulting any extra documents (e.g., course notes).
• Make sure your answers are readable. Do not use red pens or pencils.
• Please answer the exercises on the exercise sheets. If needed, also use the back sides of the exercise sheets.
• Answers on extra sheets can only be accepted if they are clearly marked with your name, your matriculation number, and the exercise number.
• Cross out text that should not be considered in the evaluation.
• Students that try to cheat do not pass the exam.
• At the end of the exam, please return all sheets together with the exercise sheets.

<table>
<thead>
<tr>
<th>Exercise</th>
<th>Total number of points</th>
<th>Number of points obtained</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exercise 1</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>Exercise 2</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>Exercise 3</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>Exercise 4</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>Exercise 5</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>58</td>
<td></td>
</tr>
<tr>
<td>Grade</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>
Exercise 1 \((4 + 5 + 4 + 6 + 5 = 24\text{ points})\)

The following data structure represents polymorphic binary trees that contain values only in special \texttt{Value} nodes that have a single successor:

\[
\text{data Tree a = Leaf | Node (Tree a) (Tree a) | Value a (Tree a)}
\]

Consider the tree \(t\) of characters on the right-hand side. The representation of \(t\) as an object of type \texttt{Tree Char} in Haskell would be:

\[(\text{Node (Value 'a' (Value 'b' Leaf)) (Node (Node Leaf Leaf) (Value 'c' Leaf)))}\]

Implement the following functions in Haskell.

(a) The function \texttt{foldTree} of type

\[
(a \rightarrow b \rightarrow b) \rightarrow (b \rightarrow b \rightarrow b) \rightarrow b \rightarrow \text{Tree a} \rightarrow b
\]

works as follows: \texttt{foldTree f g h x} replaces all occurrences of the constructor \texttt{Value} in the tree \(x\) by \(f\), it replaces all occurrences of the constructor \texttt{Node} in \(x\) by \(g\), and it replaces all occurrences of the constructor \texttt{Leaf} in \(x\) by \(h\). So for the tree \(t\) above,

\[
\text{foldTree (:) (++) [] t}
\]

should compute

\[
((++) ((:) 'a' ((:) 'b' [])) ((++) ((+) [] []) ((:) 'c' []))),
\]

which in the end results in "abc" (i.e., in the list ['a', 'b', 'c']). Here, \texttt{Value} is replaced by (:), \texttt{Node} is replaced by (++), and \texttt{Leaf} is replaced by [].


(b) Use the `foldTree` function from (a) to implement the `average` function which has the type `Tree Int -> Int` and returns the average of the values that are stored in the tree. This should be accomplished as follows:

- Use `foldTree` with suitable functions as arguments in order to compute the sum of the values stored in the trees.
- Use `foldTree` with suitable functions as arguments in order to compute the number of `Value`-objects in the tree.
- Perform integer division with the pre-defined function `div :: Int -> Int -> Int` on these values to obtain the result.

Here your function is required to work correctly only on those trees that contain the constructor `Value` at least once.
(c) Consider the following data type declaration for natural numbers:

```
data Nats = Zero | Succ Nats
```

A graphical representation of the first four levels of the domain for \( \text{Nats} \) could look like this:

```
Succ (Succ Zero)     Succ (Succ (Succ ⊥))
      |                      |
Succ Zero             Succ (Succ ⊥)
        |          |          
Zero        Succ ⊥     Succ ⊥
          |              |
              ⊥
```

Sketch a graphical representation of the first three levels of the domain for the data type \( \text{Tree Bool} \).
(d) We call a list $ys$ of integers an $n$-times even product of a list $xs$ if $ys$ has length $n$ and if all elements of $ys$ are even numbers that occur in $xs$. The goal of this exercise is to write a function $\text{evenProducts} :: [\text{Int}] \to \text{Int} \to [[\text{Int}]]$ that takes a list of integers $xs$ and a natural number $n$ and returns a list that contains all $n$-times even products of $xs$. For example, $\text{evenProducts} \ [4,5,6] \ 2 = \ [[4,4], \ [4,6], \ [6,4], \ [6,6]]$.

The following declarations are already given:

\[
\begin{align*}
\text{evenProducts} :: [\text{Int}] \to \text{Int} \to [[\text{Int}]] \\
\text{evenProducts} \ xs \ 0 & = [] \\
\text{evenProducts} \ xs \ 1 & = \text{map} \ (\lambda z \to [z]) \ (\text{filter} \ \text{even} \ xs)
\end{align*}
\]

Please give the declaration of $\text{evenProducts}$ for the missing case of numbers that are at least 2. Perform your implementation only with the help of a list comprehension, i.e., you should use exactly one declaration of the following form:

\[
\text{evenProducts} \ xs \ (n+2) = [ \ldots \ | \ldots ]
\]
(e) We define the level $n$ of a tree as the list of those values that are at distance $n$ from the root of the tree. Here, the root node has distance 0 from the root, and a non-root node has distance $n + 1$ from the root if its parent node has distance $n$ from the root.

Write a Haskell function \texttt{level :: Tree a -> Int -> [a]} which, given a tree \texttt{t} and a natural number \texttt{n}, computes the list of all values in \texttt{t} that occur there at level \texttt{n} (with repetition, i.e., a value should appear in the result list as many times as it appears on level \texttt{n}).

As an example, consider again the tree \texttt{t} from the beginning of the exercise. Here we have \texttt{level t 2 = ['b','c']} and \texttt{level t 7 = []}.
Exercise 2 \((4 + 5 = 9\) points\)

Consider the following Haskell declarations for the \texttt{fib} function, which for a natural number \(x\) computes the value \texttt{fibonacci}(x):

\[
\begin{align*}
\texttt{fib} &:: \texttt{Int} \rightarrow \texttt{Int} \\
\texttt{fib} 0 & = 0 \\
\texttt{fib} 1 & = 1 \\
\texttt{fib} (x+2) & = \texttt{fib} (x+1) + \texttt{fib} x
\end{align*}
\]

(a) Please give the Haskell declarations for the higher-order function \texttt{f_fib} corresponding to \texttt{fib}, i.e., the higher-order function \texttt{f_fib} such that the least fixpoint of \texttt{f_fib} is \texttt{fib}. In addition to the function declaration(s), please also give the type declaration of \texttt{f_fib}. Since you may use full Haskell for \texttt{f_fib}, you do not need to translate \texttt{fib} into simple Haskell.

(b) We add the Haskell declaration \texttt{bot = bot}. For each \(n \in \mathbb{N}\) please determine which function is computed by \texttt{f_fib}^{\texttt{n}} \texttt{bot}. Here \(\texttt{f_fib}^{\texttt{n}} \texttt{bot}\) represents the \(n\)-fold application of \texttt{f_fib} to \texttt{bot}, i.e., it is short for \(\texttt{f_fib} (\texttt{f_fib} \ldots (\texttt{f_fib} \texttt{bot})\ldots)^{n \text{ times}}\).

Let \(f_n : \mathbb{Z}_\bot \rightarrow \mathbb{Z}_\bot\) be the function that is computed by \(\texttt{f_fib}^{\texttt{n}} \texttt{bot}\).
Give \(f_n\) in \textbf{closed form}, i.e., using a non-recursive definition. In this definition, you may use the function \texttt{fibonacci} : \(\mathbb{N} \rightarrow \mathbb{N}\) where \(\texttt{fibonacci}(x)\) computes the \(x\)-th Fibonacci number. Here it suffices to give the result of your calculations. You do not need to present any intermediate steps.
Exercise 3 (3 + 3 = 6 points)

Let $D_1, D_2, D_3$ be domains with corresponding complete partial orders $\sqsubseteq_{D_1}, \sqsubseteq_{D_2}, \sqsubseteq_{D_3}$. As we know from the lecture, then also $\sqsubseteq_{(D_2 \times D_3)_\perp}$ is a complete partial order on $(D_2 \times D_3)_\perp$.

Now let $f : D_1 \to D_2$ and $g : D_1 \to D_3$ be functions.
We then define the function $h : D_1 \to (D_2 \times D_3)_\perp$ via $h(x) = (f(x), g(x))$.

(a) Prove or disprove: If $f$ and $g$ are strict functions, then also $h$ is a strict function.

(b) Prove or disprove: If $f$ and $g$ are monotonic functions, then also $h$ is a monotonic function.
Exercise 4 (4 + 5 = 9 points)

Consider the following data structure for polymorphic lists:

```
data List a = Nil | Cons a (List a)
```

(a) Please translate the following Haskell expression into an equivalent lambda term (e.g., using `Lam`). Recall that pre-defined functions like `odd` or `(+)` are translated into constants of the lambda calculus.

It suffices to give the result of the transformation.

```
let f = \x -> if (odd x) then (\y -> x) else f ((+) x 3)
in f
```
(b) Let $\delta$ be the set of rules for evaluating the lambda terms resulting from Haskell, i.e., $\delta$ contains at least the following rules:

\[
\begin{align*}
\text{fix} & \rightarrow \lambda f. f (\text{fix } f) \\
\text{times } 3 \ 2 & \rightarrow 6
\end{align*}
\]

Now let the lambda term $t$ be defined as follows:

\[t = (\lambda x. (\text{fix } \lambda g. x)) \ (\lambda z. (\text{times } 3 \ 2))\]

Please reduce the lambda term $t$ by WHNO-reduction with the $\rightarrow_{\beta\delta}$-relation. You have to give all intermediate steps until you reach weak head normal form (and no further steps).
Exercise 5 (10 points)

Use the type inference algorithm $W$ to determine the most general type of the following lambda term under the initial type assumption $A_0$. Show the results of all sub-computations and unifications, too. If the term is not well typed, show how and why the $W$-algorithm detects this.

$$((\text{Cons } \lambda x. x) \ y)$$

The initial type assumption $A_0$ contains at least the following:

\[
\begin{align*}
A_0(\text{Cons}) & = \forall a. (a \rightarrow (\text{List } a \rightarrow \text{List } a)) \\
A_0(x) & = \forall a. a \\
A_0(y) & = \forall a. a
\end{align*}
\]