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):

○ Master of SSE

• 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 \((4 + 5 + 4 + 5 + 6 = 24\) points\)

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

\begin{verbatim}
data Tree a = Leaf | Node (Tree a) (Tree a) | Value a (Tree a)
\end{verbatim}

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

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

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:

\[
\text{data Nats = Zero | Succ Nats}
\]

A graphical representation of the first four levels of the domain for Nats could look like this:

\[
\text{Succ (Succ Zero) \quad Succ (Succ (Succ \bot))}
\]

\[
\text{Succ Zero} \quad \text{Succ (Succ \bot)}
\]

\[
\text{Zero} \quad \text{Succ \bot}
\]

\[
\bot
\]

Sketch a graphical representation of the first three levels of the domain for the data type Tree Bool.
(d) Write a Haskell function `printStars` that first reads a string from the user, then prints this string on the console, converts the string to a number \( n \) (using the pre-defined function `read`) and in the end also prints \( n \) times the character ‘\*\*’ on the console. Also give the type declaration for your function.

You may use the `do`-notation, but you are not obliged to use it. You do not have to check whether the input string is really a number. Some of the following pre-defined functions can be helpful:

- \( \text{getLine} :: \text{IO} \, \text{String} \) reads a string from the user
- \( \text{read} :: \text{String} \rightarrow \text{Int} \) converts a string to a number
- \( \text{putStr} :: \text{String} \rightarrow \text{IO} \, () \) writes a string to the console

An example run should look as given below. Here the string “7” was read from the user.

```
Main> printStars
7
7*******
```
(e) 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 $evenProducts :: [Int] -> Int -> [[Int]]$ 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, $evenProducts [4,5,6] 2 = [[4,4], [4,6], [6,4], [6,6]]$.

The following declarations are already given:

```haskell
evenProducts :: [Int] -> Int -> [[Int]]
evenProducts xs 0 = []
evenProducts xs 1 = map (\z -> [z]) (filter even xs)
```

Please give the declaration of $evenProducts$ 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:

```haskell
evenProducts xs (n+2) = [ ... | ... ]
```
Exercise 2 (4 + 5 = 9 points)

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

\[
\text{fib} :: \text{Int} \rightarrow \text{Int} \\
\text{fib} 0 = 0 \\
\text{fib} 1 = 1 \\
\text{fib} (x+2) = \text{fib} (x+1) + \text{fib} x
\]

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

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

Let \( f_n : \mathbb{Z}_\perp \rightarrow \mathbb{Z}_\perp \) be the function that is computed by \( f \text{\_fib}^n \text{\_bot} \). Give \( f_n \) in **closed form**, i.e., using a non-recursive definition. In this definition, you may use the function \( \text{fibonacci} : \mathbb{N} \rightarrow \mathbb{N} \) where \( \text{fibonacci}(x) \) 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 \((3 + 3 = 6 \text{ points})\)

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 \(f : D_1 \rightarrow D_2\) and \(g : D_1 \rightarrow D_3\) be functions.

We then define the function \(h : D_1 \rightarrow (D_2 \times D_3)_\perp\) 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:

\[ \text{data List } a = \text{Nil} \mid \text{Cons } a \text{ (List } a) \]

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

It suffices to give the result of the transformation.

\[
\text{let } f = \lambda x \rightarrow \text{if (odd } x) \text{ then } \lambda y \rightarrow x \text{ else } f \ ((+) \ x \ 3) \\
\text{in } f
\]
(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:

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