Exam in Functional Programming SS 2014 (V3M)

First Name: ________________________________

Last Name: ________________________________

Matriculation Number: ________________________________

Course of Studies (please mark exactly one):

- Informatik Bachelor
- Informatik Master
- Mathematik Master
- Software Systems Engineering Master
- Other: ________________________________

<table>
<thead>
<tr>
<th>Exercise</th>
<th>Available Points</th>
<th>Achieved Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exercise 1</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>Exercise 2</td>
<td>29</td>
<td></td>
</tr>
<tr>
<td>Exercise 3</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>Exercise 4</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>Exercise 5</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>Sum</td>
<td>90</td>
<td></td>
</tr>
</tbody>
</table>

Notes:

- On all sheets (including additional sheets) you must write your first name, your last name and your matriculation number.
- Give your answers in readable and understandable form.
- Use permanent pens. Do not use red or green pens and do not use pencils.
- Please write your answers on the exam sheets (also use the reverse sides).
- For each exercise part, give at most one solution. Cancel out everything else. Otherwise all solutions of the particular exercise part will give you 0 points.
- If we observe any attempt of deception, the whole exam will be evaluated to 0 points.
- At the end of the exam, hand in all sheets together with the sheets containing the exam questions.
Exercise 1 (Quiz):  

(a) Give a type declaration for \( f \) such that \( (f \ True) \ (f \ 1) \) is well typed in Haskell or explain why such a type declaration cannot exist.

(b) Prove or disprove: If \( \triangleright \subseteq A \times A \) is confluent, then each \( a \in A \) has at most one normal form w.r.t. \( \triangleright \).

(c) What is the connection between monotonicity, continuity, and computability?
Exercise 2 (Programming in Haskell): \( (6 + 7 + 7 + 9 = 29 \text{ points}) \)

We define a polymorphic data structure `Tree e` for binary trees whose nodes store values of type `e`.

```haskell
data Tree e = Node e (Tree e) (Tree e) | Empty
```

The data structure `Forest e` is used to represent lists of trees.

```haskell
type Forest e = [Tree e]
```

Furthermore, we define the following data structure:

```haskell
data Animal = Squirrel | None
```

For example, `aForest` is a valid expression of type `Forest Animal`.

```haskell
aForest = [Node Squirrel Empty (Node Squirrel Empty Empty), Node None Empty Empty]
```

In this exercise, you may use full Haskell and predefined functions from the Haskell Prelude.

a) Implement a function `hunt` together with its type declaration that removes all Squirrels from a `Forest Animal`, i.e., each occurrence of a `Squirrel` should be replaced by `None`.

For example, `hunt aForest` should be evaluated to `[Node None Empty (Node None Empty Empty), Node None Empty Empty]`. 
b) Implement a function `fold :: (e -> res -> res -> res) -> res -> Tree e -> res` to fold a `Tree`.
   The first argument of `fold` is the function that is used to combine the value of the current `Node` with
   the subresults obtained for the two direct subtrees of the current `Node`. The second argument of `fold` is
   the start value, i.e., the initial subresult. The third argument is the `Tree` that has to be folded. So for
   a `Tree t`, `fold f x t` replaces the constructor `Node` by `f` and the constructor `Empty` by `x`.

   As an example, consider the following function:
   ```
   count :: Animal -> Int -> Int -> Int
   count Squirrel x y = x + y + 1
   count None x y = x + y
   ```
   Then `fold count 0 (Node Squirrel Empty (Node Squirrel Empty Empty))` should evaluate to 2,
   i.e., this application of `fold` counts all `Squirrel`s in a `Tree`.

c) Implement a function `isInhabited` together with its type declaration which gets a `Forest Animal` as
   input and returns True if and only if there is a `Tree` in the `Forest` that contains a `Squirrel`. For the
   definition of `isInhabited`, use only one defining equation where the right-hand side contains a call to
   the function `fold`. Of course, you may (and have to) use the function `fold` even if you were not able to
   solve exercise part (b). Moreover, you may use the function `count` from exercise part (b).

   Note that the function `fold` operates on a `Tree`, whereas the function `isInhabited` operates on a `Forest`!

d) In this exercise, you should implement a game where the user controls a lumberjack (i.e., a person working
   in a forest). The lumberjack walks through a `Forest Animal` and wants to cut down all `Trees` with as
   few moves as possible without damaging `Squirrels`. A move is either cutting down the current `Tree` or
   rescuing all `Squirrels` from the current `Tree` (such that it can be cut down safely, afterwards).

   Implement a function `lumberjack :: Forest Animal -> IO ()` that works as follows:
   It starts at the first `Tree` of the forest and prints "What do you want to do? (cut down (c), rescue
   squirrels (r))". If the user answers "r", then all `Squirrels` are removed from the `Tree` and the user
is asked to choose an action again. This also happens if the Tree does not contain any Squirrels. If the user answers "c" and the Tree contained Squirrels, then the function prints "You cut down a tree with squirrels!" and terminates. If the user answers "c" and the Tree does not contain Squirrels, then the function continues with the next Tree, if any, and the user is asked to choose an action again. If the user's answer is neither "r" nor "c", then the function asks to choose an action again. If no Trees are left, the function prints "You cut down all trees with n moves!" (where n is the number of moves that were performed) and terminates.

You can assume that there exists a function searchSquirrels :: Tree Animal -> Bool which checks whether there are Squirrels in a Tree and a function rescue :: Tree Animal -> Tree Animal that replaces all occurrences of Squirrel in the given Tree with None. So, for example, searchSquirrels (Node None Empty (Node Squirrel Empty Empty)) evaluates to True and rescue (Node None Empty (Node Squirrel Empty Empty)) evaluates to Node None Empty (Node None Empty Empty).

A successful run of lumberjack could look as follows:

*Main> lumberjack aForest
What do you want to do? (cut down (c), rescue squirrels (r)) r
What do you want to do? (cut down (c), rescue squirrels (r)) c
What do you want to do? (cut down (c), rescue squirrels (r)) c
You cut down all trees with 3 moves!

In the following run, the user looses the game:

*Main> lumberjack aForest
What do you want to do? (cut down (c), rescue squirrels (r)) c
You cut down a tree with squirrels!

**Hint:** You should use the function getline :: IO String to read the input from the user. To print a String, you should use the function putStrLn :: String -> IO () or the function putStrLn :: String -> IO (), if the output should end with a line break. You should use the function show :: Int -> String to convert an Int to a String. To save space, you may assume that the following declarations exist in your program:

chooseAction, lost :: String
chooseAction = "What do you want to do? (cut down (c), rescue squirrels (r)) "
lost = "You cut down a tree with squirrels!"
Exercise 3 (Semantics):
(12 + 7 + 5 = 24 points)

a) i) Let $\mathbb{N}^\infty$ be the set of all infinite sequences of natural numbers (e.g., $[0, 0, 2, 4, 4, \ldots] \in \mathbb{N}^\infty$) and let $\leq_p \subseteq \mathbb{N}^\infty \times \mathbb{N}^\infty$ be the relation that compares infinite sequences of natural numbers by their prefix sums. The $n^{th}$ prefix sum $p_n(s)$ for some $n \in \mathbb{N}$ of a sequence $s \in \mathbb{N}^\infty$ is the sum of the first $n$ elements of $s$. We have $s \leq_p s'$ if and only if $s = s'$ or there is an $n \in \mathbb{N}$ such that $p_n(s) < p_n(s')$ and $p_m(s) = p_m(s')$ for all $m \in \{0, \ldots, n-1\}$.

1) Prove that $\leq_p$ is transitive.
2) Give an example for an infinite chain in $(\mathbb{N}^\infty, \leq_p)$.
3) Prove or disprove: The partial order $\leq_p$ is complete on $\mathbb{N}^\infty$.

ii) Prove or disprove: The partial order $\leq$ is complete on $\mathbb{N}$. Here, $\leq$ is the usual "less than or equal" relation.
b) i) Consider the following Haskell function $f$:

$$
f :: \text{Int} -> \text{Int}$$

$$f 0 = 1$$

$$f x = x \times x \times f (x - 1)$$

Please give the Haskell declaration for the higher-order function $ff$ corresponding to $f$, i.e., the higher-order function $ff$ such that the least fixpoint of $ff$ is $f$. In addition to the function declaration, please also give the type declaration of $ff$. You may use full Haskell for $ff$.

ii) Let $\phi_{ff}$ be the semantics of the function $ff$. Give the least fixpoint of $\phi_{ff}$ in closed form, i.e., give a non-recursive definition of the least fixpoint of $\phi_{ff}$.

**Hint:** For natural numbers $x$, the factorial function can be defined as follows:

$$x! = \begin{cases} 
1 & \text{if } x = 0 \\
 x \cdot (x-1)! & \text{if } x > 0 
\end{cases}$$
c) Consider the data type declarations on the left and, as an example, the graphical representation of the first three levels of the domain for \texttt{Nats} on the right:

\begin{center}
\begin{tabular}{|c|c|}
\hline
\texttt{data Nats} & \begin{tabular}{c}
\texttt{S Z} \\
\texttt{S (S ⊥)}
\end{tabular} \\
\hline
\texttt{type Forest e} & \begin{tabular}{c}
\texttt{Z} \\
\texttt{S ⊥}
\end{tabular} \\
\hline
\texttt{data Tree e} & \begin{tabular}{c}
\texttt{⊥}
\end{tabular} \\
\hline
\end{tabular}
\end{center}

Give a graphical representation of the first three levels of the domain for the type \texttt{Forest Int}. The third level contains the element \texttt{Empty}: \texttt{⊥}, for example.
Exercise 4 (Lambda Calculus): \(4 + 8 + 6 = 18\) points

a) Reconsider the function \(f\) from the previous exercise:

\[
\begin{align*}
  f : & \mathbb{Int} \rightarrow \mathbb{Int} \\
  f(x) &= \begin{cases} 
  1 & \text{if } x = 0 \\
  x \times x \times f(x - 1) & \text{otherwise}
  \end{cases}
\end{align*}
\]

Please implement this function in the Lambda Calculus, i.e., give a term \(t\) such that, for all \(x, y \in \mathbb{Z}\),

\[f(x) = y\] if and only if \(t\) can be reduced to \(y\) via WHNO-reduction with the \(\rightarrow_{\beta\delta}\)-relation and the set of rules \(\delta\) as introduced in the lecture to implement Haskell. You can use infix notation for predefined functions like \((=), (\times)\) or \((-)\).

b) Let \(t = \lambda g \:\text{if } (x == 0) x (g x)\) and

\[
\delta = \begin{cases} 
  \text{if True} & \rightarrow \lambda x y. x, \\
  \text{if False} & \rightarrow \lambda x y. y, \\
  \text{fix} & \rightarrow \lambda f. f(\text{fix } f)
\end{cases}
\]

\[
\cup \{ x == x \rightarrow \text{True} \mid x \in \mathbb{Z} \} \\
\cup \{ x == y \rightarrow \text{False} \mid x, y \in \mathbb{Z}, x \neq y \}
\]

Please reduce \(\text{fix } 0\) by WHNO-reduction with the \(\rightarrow_{\beta\delta}\)-relation. List all intermediate steps until reaching weak head normal form, but please write “\(t\)” instead of \(\lambda g \:\text{if } (x == 0) x (g x)\) whenever possible.
c) Consider the Boolean operator \texttt{nand} where \texttt{nand}(x,y) holds if and only if \texttt{and}(x,y) does not hold. Using the representation of Boolean values in the pure $\lambda$-calculus presented in the lecture, i.e., $\texttt{True}$ is represented as $\lambda x \ y. x$ and $\texttt{False}$ as $\lambda x \ y. y$, give a pure $\lambda$-term for \texttt{nand} in $\rightarrow_\beta$-normal form.

In your solution, you may abbreviate the $\lambda$-term $\lambda x \ y. x$ with $\texttt{True}$ and the $\lambda$-term $\lambda x \ y. y$ with $\texttt{False}$. 
Exercise 5 (Type Inference):  

Using the initial type assumption $A_0 := \{x :: \forall a.a, g :: \forall a.a\}$, infer the type of the expression $\lambda f.g(fx)$ using the algorithm $W$.  

(10 points)