Exam in Functional Programming SS 2014 (V3B)

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>20</td>
<td></td>
</tr>
<tr>
<td>Exercise 3</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>Exercise 4</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>Exercise 5</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>Exercise 6</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): (3 + 3 + 3 = 9 points)

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 = 20 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 `Squirrel`s 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:

```haskell
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`!
Exercise 3 (List Comprehensions): \(4 + 7 = 11\) points

In this exercise, you can assume that there exists a function \(\text{divisors} :: \text{Int} \to \text{[Int]}\) where, for any natural number \(x \geq 2\), \(\text{divisors} x\) computes the list of all its proper divisors (including 1, but excluding \(x\)). So for example, \(\text{divisors} 6 = [1,2,3]\).

a) Write a Haskell expression in form of a list comprehension to compute all amicable pairs of numbers. A pair of natural numbers \((x,y)\) with \(x > y \geq 2\) is amicable if and only if the sum of the proper divisors of \(x\) is equal to \(y\) and the sum of the proper divisors of \(y\) is equal to \(x\). For example, \((284, 220)\) is amicable:

- The proper divisors of 284 are 1, 2, 4, 71, and 142 and their sum is 220.
- The proper divisors of 220 are 1, 2, 4, 5, 10, 11, 20, 22, 44, 55, and 110 and their sum is 284.

In other words, give a list comprehension for a list that only contains amicable pairs of numbers and, for every amicable pair of numbers \(p\), there is an \(n \in \mathbb{N}\) such that the \(n^{th}\) element of the list is \(p\).

**Hint:** The function \(\text{sum} :: \text{[Int]} \to \text{Int}\) computes the sum of a list of integers.

b) Write a Haskell expression in form of a list comprehension to compute all practical numbers. A natural number \(x \geq 2\) is practical if and only if each smaller number \(y \in \{1, \ldots, x - 1\}\) is equal to the sum of some of \(x\)'s proper divisors. For example, 6 is practical: Its proper divisors are 1, 2, and 3 and we have \(4 = 3 + 1\) and \(5 = 3 + 2\).

In your solution, you may use the function \(\text{sum}\) and the following functions:

- The function \(\text{any} :: (a \to \text{Bool}) \to \text{[a]} \to \text{Bool}\) tests whether there is an element in the given list that satisfies the given predicate.
- The function \(\text{all} :: (a \to \text{Bool}) \to \text{[a]} \to \text{Bool}\) tests whether all elements in the given list satisfy the given predicate.
- The function \(\text{subsequences}[a] \to [[a]]\) computes all subsequences of the given list. For example, we have:

\[
\text{subsequences } [1,2,3] = [[], [1], [2], [1,2], [3], [1,3], [2,3], [1,2,3]]
\]
Exercise 4 (Semantics): \[12 + 7 + 5 + 4 = 28\] 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 \( f \):

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

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{verbatim}
data Nats = Z | S Nats

type Forest e = [Tree e]
data Tree e =
     Node e (Tree e) (Tree e)
     | Empty
\end{verbatim}

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}: \(\bot\), for example.
d) Reconsider the definition for Nats from the previous exercise part, i.e.,
\[\text{data Nats = Z | S Nats}\]. Moreover, reconsider the function \(f\):
\[
f :: \text{Int} \to \text{Int}
f 0 = 1
f x = x \times x \times f (x - 1)
\]
Write a function \(\text{fNat} :: \text{Nats} \to \text{Nats}\) in Simple Haskell which, for natural numbers, computes the same result as the function \(f\). That means, if \(n \geq 0\) and \(f n = x\), then we have \(\text{fNat} (\text{S}^n \text{Z}) = \text{S}^x \text{Z}\).
You can assume that there exists a predefined function \(\text{mult} :: \text{Nats} \to \text{Nats} \to \text{Nats}\) to multiply two natural numbers. However, there is no predefined function to subtract natural numbers of type \(\text{Nats}\).
Your solution should use the functions defined in the transformation from the lecture such as \(\text{isa}_{\text{constr}}\) and \(\text{argof}_{\text{constr}}\). You do not have to use the transformation rules from the lecture, though.
Exercise 5 (Lambda Calculus): 

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

\[
f :: \text{Int} \rightarrow \text{Int} \\
f 0 = 1 \\
f x = x * x * f (x - 1)
\]

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 \( t x \) 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 \((==)\), \((\ast)\) or \((\neg)\).

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

\[
\delta = \{ \text{if True} \rightarrow \lambda x \ y. x, \\
\text{if False} \rightarrow \lambda x \ y. y, \\
\text{fix} \rightarrow \lambda f. f(\text{fix} f) \} \\
\cup \{ x == x \rightarrow \text{True} \mid x \in \mathbb{Z} \} \\
\cup \{ x == y \rightarrow \text{False} \mid x, y \in \mathbb{Z}, x \neq y \}
\]

Please reduce \( \text{fix} \, t \, 0 \) 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 \( \lambda g \ x. \text{if} \ (x == 0) \ x \ (g \ x) \) whenever possible.
Exercise 6 (Type Inference): (10 points)

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