Exercise 1 (Quiz): \(3 + 3 + 3 = 9\) points

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

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

c) What is the connection between monotonicity, continuity, and computability?

Solution:

a) \(f :: a \rightarrow b \rightarrow \text{Bool}\)

b) Let \(q_1\) and \(q_2\) be \(\succ\)-normal forms of \(t\). Then \(t \succ^* q_1\) and \(t \succ^* q_2\). By confluence of \(\succ\), there must be a \(q\) such that \(q_1 \succ^* q\) and \(q_2 \succ^* q\). Since \(q_1\) and \(q_2\) are normal forms, we get \(q_1 = q = q_2\).

c) Every computable function is continuous and every continuous function is monotonic.

Exercise 2 (Programming in Haskell): \(6 + 7 + 7 = 20\) points

We define a polymorphic data structure \(\text{Tree}\; e\) for binary trees whose nodes store values of type \(e\).

\[
\text{data Tree}\; e = \text{Node}\; e\; (\text{Tree}\; e)\; (\text{Tree}\; e)\; |\; \text{Empty}
\]

The data structure \(\text{Forest}\; e\) is used to represent lists of trees.

\[
\text{type Forest}\; e = [\text{Tree}\; e]
\]

Furthermore, we define the following data structure:

\[
\text{data Animal} = \text{Squirrel} | \text{None}
\]

For example, \(\text{aForest}\) is a valid expression of type \(\text{Forest}\; \text{Animal}\).

\[
\text{aForest} = [\text{Node}\; \text{Squirrel}\; \text{Empty}\; (\text{Node}\; \text{Squirrel}\; \text{Empty}\; \text{Empty}), \text{Node}\; \text{None}\; \text{Empty}\; \text{Empty}]
\]

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

a) Implement a function \(\text{hunt}\) together with its type declaration that removes all \(\text{Squirrel}\)s from a \(\text{Forest}\; \text{Animal}\), i.e., each occurrence of a \(\text{Squirrel}\) should be replaced by \(\text{None}\).

For example, \(\text{hunt}\; \text{aForest}\) should be evaluated to \([\text{Node}\; \text{None}\; \text{Empty}\; (\text{Node}\; \text{None}\; \text{Empty}\; \text{Empty}), \text{Node}\; \text{None}\; \text{Empty}\; \text{Empty}].

b) Implement a function \(\text{fold} :: (e \rightarrow \text{res} \rightarrow \text{res} \rightarrow \text{res}) \rightarrow \text{res} \rightarrow \text{Tree}\; e \rightarrow \text{res}\) to fold a \(\text{Tree}\). The first argument of \(\text{fold}\) is the function that is used to combine the value of the current \(\text{Node}\) with the subresults obtained for the two direct subtrees of the current \(\text{Node}\). The second argument of \(\text{fold}\) is the start value, i.e., the initial subresult. The third argument is the \(\text{Tree}\) that has to be folded. So for a \(\text{Tree}\; t\), \(\text{fold}\; f\; x\; t\) replaces the constructor \(\text{Node}\) by \(f\) and the constructor \(\text{Empty}\) by \(x\).

As an example, consider the following function:

\[
\text{count} :: \text{Animal} \rightarrow \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}
\]
\[
\text{count}\; \text{Squirrel}\; x\; y = x + y + 1
\]
\[
\text{count}\; \text{None}\; x\; y = x + y
\]
Then fold count 0 (Node Squirrel Empty (Node Squirrel Empty Empty)) should evaluate to 2, i.e., this application of fold counts all Squirrels 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!

Solution:

\[
\text{a)} \quad \text{hunt} :: \text{Forest Animal} \rightarrow \text{Forest Animal} \\
\text{hunt forest} = \text{map} \ f \ \text{forest} \ where \\
f \ \text{Empty} = \text{Empty} \\
f \ (\text{Node} \ _ \ 1 \ r) = \text{Node} \ \text{None} \ (f \ 1) \ (f \ r) \\
\text{b)} \quad \text{fold} :: (e \rightarrow \text{res} \rightarrow \text{res} \rightarrow \text{res}) \rightarrow \text{res} \rightarrow \text{Tree} \ e \rightarrow \text{res} \\
\text{fold} \ f \ x \ \text{Empty} = x \\
\text{fold} \ f \ x \ (\text{Node} \ v \ l \ r) = f \ v \ (\text{fold} \ f \ x \ l) \ (\text{fold} \ f \ x \ r) \\
\text{c)} \quad \text{isInhabited} :: \text{Forest Animal} \rightarrow \text{Bool} \\
\text{isInhabited} \ \text{xs} = \text{sum} \ (\text{fold} \ \text{count} \ 0 \ \text{xs}) > 0
\]

Exercise 3 (List Comprehensions): (4 + 7 = 11 points)

In this exercise, you can assume that there exists a function \text{divisors} :: \text{Int} \rightarrow [\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 \(p\).

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

\[
\text{subsequences} \ [1,2,3] = [[],[1],[2],[1,2],[3],[1,3],[2,3],[1,2,3]]
\]
Solution:

a) \[(x,y) \mid x \leftarrow [3..], y \leftarrow [2..x-1], \text{sum (divisors } x) = y, \text{sum (divisors } y) = x\]

b) \[x \mid x \leftarrow [2..], \text{all (} y \rightarrow \text{any (} s \rightarrow \text{sum } s = y\text{) (subsequences (divisors } x))\text{) [2..x-1]}\]

Exercise 4 (Semantics):

12 + 7 + 5 + 4 = 28 points

a)  

i) Let \(\mathbb{N}\) be the set of all infinite sequences of natural numbers (e.g., \([0, 0, 2, 2, 4, 4, \ldots] \in \mathbb{N}\)) and let \(\leq_p \subseteq \mathbb{N} \times \mathbb{N}\) 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 \(s \in \mathbb{N}\) 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 \((\mathbb{N}, \leq_p)\).

3) Prove or disprove: The partial order \(\leq_p\) is complete on \(\mathbb{N}\).

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 \(f\):

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

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 fixed point 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 fixed point of \(\phi_{ff}\) in closed form, i.e., give a non-recursive definition of the least fixed point 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 \(\text{Nats}\) on the right:

<table>
<thead>
<tr>
<th>data Nats = Z</th>
<th>S Nats</th>
<th>S Z</th>
<th>S (S ⊥)</th>
<th>3rd level</th>
</tr>
</thead>
<tbody>
<tr>
<td>type Forest e = [Tree e]</td>
<td></td>
<td>Z</td>
<td>S ⊥</td>
<td>2nd level</td>
</tr>
<tr>
<td>data a Tree e =</td>
<td></td>
<td></td>
<td></td>
<td>1st level</td>
</tr>
<tr>
<td>Node e (Tree e) (Tree e)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Empty</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Give a graphical representation of the first three levels of the domain for the type Forest Int. The third level contains the element Empty: ⊥, for example.

d) Reconsider the definition for Nat from the previous exercise part, i.e., data Nat = Z | S Nat. Moreover, reconsider the function f:

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

Write a function fNat :: Nat \to Nat 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 fNat (S^n Z) = S^n Z. You can assume that there exists a predefined function mult :: Nat \to Nat to multiply two natural numbers. However, there is no predefined function to subtract natural numbers of type Nat.

Your solution should use the functions defined in the transformation from the lecture such as isa and argo of constr. You do not have to use the transformation rules from the lecture, though.

Solution:

a) i) 1) Let x \leq_p y \leq_p z. If x = y or y = z, we are done. Otherwise, there is an n \in \mathbb{N} such that p_n(x) < p_n(y) and p_m(x) = p_m(y) for all m \in \{1, \ldots, n - 1\} and an n' \in \mathbb{N} such that p_{n'}(y) < p_{n'}(z) and p_m(y) = p_m(z) for all m \in \{1, \ldots, n' - 1\}. Let n'' = min(n, n'). Then p_{n''}(x) < p_{n''}(z) and p_m(x) \leq p_m(z) for all m \in \{1, \ldots, n'' - 1\} and thus x \leq_p z.

2) \{[0, 0, 0, \ldots], [1, 0, 0, \ldots], [2, 0, 0, \ldots], \ldots\}

3) Consider the chain above. Its least upper bound is [\infty, 0, 0, \ldots] \notin \mathbb{N}^\infty. Thus, \leq_p is not a cpo on \mathbb{N}^\infty.

ii) Consider the chain \mathbb{N}. Its least upper bound is \infty \notin \mathbb{N}. Thus, \leq is not complete on \mathbb{N}.

b) i) ff :: (Int -> Int) \to (Int -> Int)

\[
\begin{align*}
    ff\ f\ 0 & =\ 1 \\
    ff\ f\ x & =\ x \times x \times f\ (x - 1)
\end{align*}
\]

ii) \[\text{lfp } \phi_{x}\](x) = \begin{cases} 
    (x!)^2 & \text{if } 0 \leq x \\
    \bot & \text{otherwise}
\end{cases}

c)

\[
\begin{align*}
    \text{Node } \bot \bot \bot & : \bot \\
    \text{Empty} & : \bot \\
    \bot : & [] \\
    \bot : (\bot : \bot)
\end{align*}
\]
d) \( f_{\text{Nat}} = \lambda x \rightarrow \)
   \begin{align*}
   &\text{if } (\text{isa}_Z \ x) \text{ then } 1 \\
   &\text{else } \text{mult} \ x \ (\text{mult} \ x \ (f_{\text{Nat}} \ (\text{argof}_S \ x)))
   \end{align*}

Exercise 5 (Lambda Calculus): (4 + 8 = 12 points)

a) Reconsider the function \( f \) from the previous exercise:
   \begin{align*}
   f :: \text{Int} \rightarrow \text{Int} \\
   f \ 0 &= 1 \\
   f \ x &= x * x * f \ (x - 1)
   \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 \( t \ x \) 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 \((=), \(*)\) or \((-)\).

b) Let \( t = \lambda g \ . \ x \ . \ (x == 0) \ x \ (g \ x) \) and
   \begin{align*}
   \delta &= \{ \text{if True } \rightarrow \lambda x \ y \ . \ x, \\
   &\text{if False } \rightarrow \lambda x \ y \ . \ y, \\
   &\text{fix } \rightarrow \lambda f \ . (\text{fix} \ f) \}
   \end{align*}

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 \( \lambda g \ . \ x \ . \ (x == 0) \ x \ (g \ x) \) whenever possible.

Solution:

\[ \text{a) } \text{fix} \ (\lambda f \ . \ x \ . \ (x == 0) \ x \ * \ x \ (f \ (x - 1))) \]

\[ \text{b) } \]

\begin{align*}
\text{fix} \ t \ 0 &\rightarrow_\beta (\lambda f \ . \ (f \ (\text{fix} \ f))) \ t \ 0 \\
&\rightarrow_\beta t \ ((\text{fix} \ t) \ 0 \\
&\rightarrow_\beta (\lambda x \ . \ (x == 0) \ x \ (\text{fix} \ t \ x)) \ 0 \\
&\rightarrow_\beta \text{if} \ (0 == 0) \ 0 \ ((\text{fix} \ t) \ 0) \\
&\rightarrow_\beta \text{if} \ True \ 0 \ ((\text{fix} \ t) \ 0) \\
&\rightarrow_\beta (\lambda x \ y \ . \ x) \ 0 \ ((\text{fix} \ t) \ 0) \\
&\rightarrow_\beta (\lambda y \ 0) \ ((\text{fix} \ t) \ 0) \\
&\rightarrow_\beta 0
\end{align*}
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 $W$.

Solution:

$$W(A_0, \lambda f. g \, (f \, x))$$

$$W(A_0 + \{ f :: b_1 \}, g \, (f \, x))$$

$$W(A_0 + \{ f :: b_1 \}, g) = (id, b_2)$$

$$W(A_0 + \{ f :: b_1 \}, f)$$

$$W(A_0 + \{ f :: b_1 \}, x) = (id, b_3)$$

$$mgu(b_1, b_3 \rightarrow b_4) = [b_1/b_3 \rightarrow b_4]$$

$$= ([b_1/b_3 \rightarrow b_4], b_2)$$

$$mgu(b_2, b_4 \rightarrow b_5) = [b_2/b_4 \rightarrow b_5]$$

$$= ([b_1/b_3 \rightarrow b_4, b_2/b_4 \rightarrow b_5], b_5)$$

$$= ([b_1/b_3 \rightarrow b_4, b_2/b_4 \rightarrow b_5], (b_3 \rightarrow b_4) \rightarrow b_5)$$