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

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

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

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

Solution:

\[ a) f :: a -> b -> Bool \]

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

\[ c) \text{Every computable function is continuous and every continuous function is monotonic.} \]

Exercise 2 (Programming in Haskell):  
(6 + 7 + 7 + 9 = 29 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:

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

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

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

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

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

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

\[ b) \text{Implement a function } \text{fold} :: (e -> \text{res} -> \text{res} -> \text{res}) -> \text{res} -> \text{Tree } e -> \text{res} \text{ to fold a Tree.} \]

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 \( \text{Tree } t, \text{fold } f \times t \) replaces the constructor \( \text{Node} \) by \( f \) and the constructor \( \text{Empty} \) by \( x \).

As an example, consider the following function:

\[ \text{count} :: \text{Animal} -> \text{Int} -> \text{Int} -> \text{Int} \]

\[ \text{count Squirrel } x \ y = x + y + 1 \]

\[ \text{count None } x \ y = x + y \]
Then fold count 0 (Node Squirrel Empty (Node Squirrel Empty Empty)) should evaluate to 2, i.e., this application of fold counts all Squirrels in a Tree.

e) 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
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!"

Solution: 2
a) hunt :: Forest Animal -> Forest Animal
hunt forest = map f forest where
  f Empty = Empty
  f (Node _ l r) = Node None (f l) (f r)

b) fold :: (e -> res -> res -> res) -> res -> Tree e -> res
fold f x Empty = x
fold f x (Node v l r) = f v (fold f x l) (fold f x r)

c) isInhabited :: Forest Animal -> Bool
isInhabited xs = sum (fold count 0 xs) > 0

d) lumberjack :: Forest Animals -> IO()
lumberjack trees = lumberjack' 0 trees where
  lumberjack' moves [] = putStrLn ("You cut down all trees with " ++ (show moves) ++ " moves!")
  lumberjack' moves (x:xs) = do
    putStr chooseAction
    a <- getLine
    case a of "c" -> if searchSquirrels x then putStrLn lost
      else lumberjack' (moves+1) xs
    "r" -> lumberjack' (moves+1) (rescueSquirrels x:xs)
    _ -> lumberjack' moves (x:xs)

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., \([0, 0, 2, 2, 4, 4, \ldots] \in \mathbb{N}^\infty \)) 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 \( f \):

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

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 \( \text{Nats} \) on the right:
data Nats = Z | S Nats

\[
\begin{array}{ccc}
S Z & S (S \bot) & 3^{rd} \text{ level} \\
\downarrow & & \\
Z & S \bot & 2^{nd} \text{ level} \\
\downarrow & & \\
\bot & & 1^{st} \text{ level}
\end{array}
\]

Give a graphical representation of the first three levels of the domain for the type Forest Int. The third level contains the element Empty: \(\bot\), for example.

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 \(p_n(x) = p_n(y)\) for all \(m \in \{1, \ldots, n - 1\}\) and an \(n' \in \mathbb{N}\) such that \(p_{n'}(y) < p_{n'}(z)\) and \(p_{n'}(y) = p_{n'}(z)\) for all \(m \in \{1, \ldots, n' - 1\}\). Let \(n'' = \min(n, n')\). Then \(p_{n''}(x) < p_{n''}(z)\) and \(p_{n''}(x) \leq p_{n''}(z)\) 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) \(\text{ff} :: (\text{Int} \to \text{Int}) \to (\text{Int} \to \text{Int})\)

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

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

c)

\[
\begin{array}{ccc}
(\text{Node } \bot \bot \bot) : \bot & \text{Empty} : \bot & \bot : [] \\
\downarrow & & \downarrow : (\bot : \bot)
\end{array}
\]

[]

\[
\begin{array}{ccc}
\bot : \bot & \downarrow : \bot & \bot
\end{array}
\]

\[
\bot
\]
Exercise 4 (Lambda Calculus): \(4 + 8 + 6 = 18\) points

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

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

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 \((==), \ast\) or \((-)\).

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

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

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

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.

c) Consider the Boolean operator \(\text{nand}\) where \(\text{nand}(x, y)\) holds if and only if \(\text{and}(x, y)\) does not hold. Using the representation of Boolean values in the pure \(\lambda\)-calculus presented in the lecture, i.e., \(\text{True}\) is represented as \(\lambda x y. x\) and \(\text{False}\) as \(\lambda x y. y\), give a pure \(\lambda\)-term for \(\text{nand}\) in \(\rightarrow_{\beta}\)-normal form.

In your solution, you may abbreviate the \(\lambda\)-term \(\lambda x y. x\) with \(\text{True}\) and the \(\lambda\)-term \(\lambda x y. y\) with \(\text{False}\).

Solution:

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

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

c) \(\text{nand} = \lambda x y. x \ (\lambda x y. y) \ x \ (\lambda x y. x)\)
Exercise 5 (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 \( \mathcal{W} \).

Solution:

\[
\begin{align*}
\mathcal{W}(A_0, \lambda f. g(f x)) \\
\mathcal{W}(A_0 + \{ f :: b_1 \}, g(f x)) \\
\mathcal{W}(A_0 + \{ f :: b_1 \}, g) = (id, b_2) \\
\mathcal{W}(A_0 + \{ f :: b_1 \}, f x) \\
\mathcal{W}(A_0 + \{ f :: b_1 \}, f) = (id, b_1) \\
\mathcal{W}(A_0 + \{ f :: b_1 \}, x) = (id, b_3) \\
mgu(b_1, b_3 \rightarrow b_4) = [b_1/b_3 \rightarrow b_4] \\
= ([b_1/b_3 \rightarrow b_4], b_4) \\
mgu(b_2, b_4 \rightarrow b_5) = [b_2/b_4 \rightarrow b_5] \\
= ([b_1/b_3 \rightarrow b_4, b_2/b_4 \rightarrow b_5], b_5) \\
= ([b_1/b_3 \rightarrow b_4, b_2/b_4 \rightarrow b_5], (b_3 \rightarrow b_4) \rightarrow b_5)
\end{align*}
\]