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 $\succ \subseteq A \times A$ 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 -> b -> 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 + 9 = 29 points)

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

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

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

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

Furthermore, we define the following data structure:

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

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

```
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:

```
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 `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`!

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 `putStr :: 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:

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`:

```
f :: Int -> Int
f 0 = 1
f x = x * x * 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 $\phi_{\texttt{ff}}$ be the semantics of the function `ff`. Give the least fixpoint of $\phi_{\texttt{ff}}$ in closed form, i.e., give a non-recursive definition of the least fixpoint of $\phi_{\texttt{ff}}$.

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 `Nats` on the right:

```
data Nats = Z | S Nats

--------------------------------------

type Forest e = [Tree e]
data Tree e =
    Node e (Tree e) (Tree e)
    | Empty
```

$$
\begin{tikzcd}
 & \texttt{S Z} & & \texttt{S (S }\bot\texttt{)} & 3^{rd}\text{ level} \\
\texttt{Z} & & \texttt{S }\bot \arrow[ul] \arrow[ur] & & 2^{nd}\text{ level} \\
 & \bot \arrow[ul] \arrow[ur] & & & 1^{st}\text{ level}
\end{tikzcd}
$$

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 $p_m(x) = p_m(y)$ for all $m \in \{1, \dots, n-1\}$ and an $n' \in \mathbb{N}$ such that $p_{n'}(y) < p_{n'}(z)$ and $p_m(y) = p_m(z)$ for all $m \in \{1, \dots, n'-1\}$. Let $n'' = min(n, n')$. Then $p_{n''}(x) < p_{n''}(z)$ and $p_m(x) \leq p_m(z)$ for all $m \in \{1, \dots, n''-1\}$ and thus $x \leq_p z$.

2) $\{[0,0,0,\dots],[1,0,0,\dots],[2,0,0,\dots],\dots\}$

3) Consider the chain above. Its least upper bound is $[\infty,0,0,\dots] \notin \mathbb{N}^\infty$. 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) -> (Int -> Int)
ff f 0 = 1
ff f x = x * x * f (x - 1)
```

ii)

$$(\mathsf{lfp}\ \phi_{\texttt{ff}})(x) = \begin{cases} (x!)^2 & \text{if } 0 \leq x \\ \bot & \text{otherwise} \end{cases}$$

c)

$$
\begin{tikzcd}
(\texttt{Node}\ \bot\ \bot\ \bot) : \bot & & \texttt{Empty} : \bot & & \bot : [] & & \bot : (\bot : \bot) \\
 & [] & & \bot : \bot \arrow[ulll] \arrow[ul] \arrow[ur] \arrow[urrr] & & & \\
 & & \bot \arrow[ul] \arrow[ur] & & & &
\end{tikzcd}
$$

Exercise 4 (Lambda Calculus): (4 + 8 + 6 = 18 points)

a) Reconsider the function `f` from the previous exercise:

```
f :: Int -> 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 $\mathtt{x}, \mathtt{y} \in \mathbb{Z}$, `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 δ as introduced in the lecture to implement Haskell. You can use infix notation for predefined functions like $(==)$, $(*)$ or $(-)$.

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

$$\begin{aligned}\delta = \ & \{\ \mathtt{if}\ \mathtt{True} \rightarrow \lambda\, x\ y.\, x, \\ & \quad \mathtt{if}\ \mathtt{False} \rightarrow \lambda\, x\ y.\, y, \\ & \quad \mathtt{fix} \rightarrow \lambda\, f.\, f(\mathtt{fix}\ f)\} \\ \cup & \ \{\ x == x \rightarrow \mathtt{True} \mid x \in \mathbb{Z}\} \\ \cup & \ \{\ x == y \rightarrow \mathtt{False} \mid x, y \in \mathbb{Z}, x \neq y\}\end{aligned}$$

Please reduce $\mathtt{fix}\,\mathtt{t}\,0$ 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 $\lambda g\ x.\mathtt{if}\ (x == 0)\ x\ (g\ x)$ whenever possible.

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

In your solution, you may abbreviate the λ-term $\lambda\, x\ y.\, x$ with `True` and the λ-term $\lambda\, x\ y.\, y$ with `False`.

Solution:

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

b)

$$\begin{aligned} & \mathtt{fix}\ \mathtt{t}\ 0 \\ \rightarrow_\delta\ & (\lambda\, f.\, (f\ (\mathtt{fix}\ f)))\ \mathtt{t}\ 0 \\ \rightarrow_\beta\ & \mathtt{t}\ (\mathtt{fix}\ \mathtt{t})\ 0 \\ \rightarrow_\beta\ & (\lambda\, x.\, \mathtt{if}\ (x == 0)\ x\ (\mathtt{fix}\ \mathtt{t}\ x))\ 0 \\ \rightarrow_\beta\ & \mathtt{if}\ (0 == 0)\ 0\ (\mathtt{fix}\ \mathtt{t}\ 0) \\ \rightarrow_\delta\ & \mathtt{if}\ \mathtt{True}\ 0\ (\mathtt{fix}\ \mathtt{t}\ 0) \\ \rightarrow_\delta\ & (\lambda\, x\ y.\, x)\ 0\ (\mathtt{fix}\ \mathtt{t}\ 0) \\ \rightarrow_\beta\ & (\lambda\, y.\, 0)\ (\mathtt{fix}\ \mathtt{t}\ 0) \\ \rightarrow_\beta\ & 0 \end{aligned}$$

c) $\mathtt{nand} = \lambda x\ y.x\ (y\ (\lambda x\ y.y)\ x)\ (\lambda x\ y.x)$

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:

$$
\begin{array}{l}
\mathcal{W}(A_0, \lambda f.g\,(f\,x)) \\
\quad \mathcal{W}(A_0 + \{f :: b_1\}, g\,(f\,x)) \\
\quad\quad \mathcal{W}(A_0 + \{f :: b_1\}, g) = (id, b_2) \\
\quad\quad \mathcal{W}(A_0 + \{f :: b_1\}, f\,x) \\
\quad\quad\quad \mathcal{W}(A_0 + \{f :: b_1\}, f) = (id, b_1) \\
\quad\quad\quad \mathcal{W}(A_0 + \{f :: b_1\}, x) = (id, b_3) \\
\quad\quad mgu(b_1, b_3 \to b_4) = [b_1/b_3 \to b_4] \\
\quad\quad = ([b_1/b_3 \to b_4], b_4) \\
\quad mgu(b_2, b_4 \to b_5) = [b_2/b_4 \to b_5] \\
\quad = ([b_1/b_3 \to b_4, b_2/b_4 \to b_5], b_5) \\
= ([b_1/b_3 \to b_4, b_2/b_4 \to b_5], (b_3 \to b_4) \to b_5)
\end{array}
$$