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

a) Is \(\lambda x . (f \ True) (f \ 1)\) well typed in Haskell? Give the expression’s type or briefly explain why it is not well typed.

b) Prove or disprove: If a relation \(\triangleright\subseteq A \times A\) is confluent, then every element of \(A\) has a normal form with respect to \(\triangleright\).

c) Are there monotonic functions which are not continuous? If so, give an example. Otherwise, give a brief explanation.

Solution:

a) No, because the most general type schema for this Haskell expression is non-flat, but such type schemata are not allowed in Haskell.

b) Counterexample: \(A = \{a\}\) with \(a \triangleright a\). Obviously, the relation is confluent and \(a\) does not have a normal form w.r.t. \(\triangleright\).

c) Yes, e.g., the function \(g : (\mathbb{Z}_\perp \to \mathbb{Z}_\perp) \to \mathbb{Z}_\perp\) defined by

\[
g(f) = \begin{cases} 
0, & \text{if } f(x) \neq \perp \text{ for all } x \in \mathbb{Z} \\
\perp \mathbb{Z}_\perp, & \text{otherwise}
\end{cases}
\]

is monotonic, but not continuous.

Exercise 2 (Programming in Haskell): \(5 + 7 + 7 + 9 = 28\) points

We define a polymorphic data structure \texttt{Train} to represent trains that can contain different types of cargo.

```haskell
data Train a = Locomotive (Train a) | Wagon a (Train a) | Empty deriving Show
data Cargo = NoCargo | Persons Quantity | Goods Weight deriving Show
```

The data structure \texttt{Cargo} is used to represent different types of cargo.

```haskell
type Quantity = Int

type Weight = Int -- in kg
```

For example, \texttt{aTrain} is a valid expression of type \texttt{Train Cargo}.

```haskell
aTrain = Locomotive (Wagon (Goods 100) (Wagon (Persons 10) (Wagon (Goods 200) Empty)))
```

Like \texttt{aTrain}, you can assume that every \texttt{Train} consists of a single \texttt{Locomotive} at its beginning followed by a sequence of \texttt{Wagons} and \texttt{Empty} at its end.

The following function can be used to \textit{fold} a \texttt{Train}. 
fold :: (a -> b -> b) -> b -> Train a -> b
fold _ res Empty = res
fold f res (Locomotive t) = fold f res t
fold f res (Wagon c t) = f c (fold f res t)

So for a Train t, fold f res t removes the constructor Locomotive, replaces Wagon by f, and replaces Empty by res.

In the following exercises, you are allowed to use predefined functions from the Haskell-Prelude.

a) Implement a function filterTrain together with its type declaration (filterTrain :: ...). The function filterTrain gets a predicate and an object of type Train a as input and returns an object of type Train a that only contains those wagons from the given Train whose cargo satisfies the predicate.

For example, assume that the function areGoods is implemented as follows:

areGoods :: Cargo -> Bool
areGoods (Goods _) = True
areGoods _ = False

Then the expression filterTrain areGoods aTrain should be evaluated to Locomotive (Wagon (Goods 100) (Wagon (Goods 200) Empty)).

b) Implement a function buildTrain :: [Cargo] -> Train Cargo. In the resulting Train, a single Wagon must not contain more than 1000 kg of Goods. If the input list contains Goods that weigh more than 1000 kg, then these Goods must not be contained in the resulting train. Apart from this restriction, all the Cargo given via the input list has to be contained. Moreover, the resulting Train has to consist of a single Locomotive at its beginning, followed by a sequence of Wagons and Empty at its end. In your solution, you should use the function filterTrain even if you could not solve the previous exercise part.

For example, buildTrain [Persons 10, Goods 2000, Goods 1000] should be evaluated to the expression Locomotive (Wagon (Persons 10) (Wagon (Goods 1000) (Wagon (Goods 200) Empty))).

c) Implement a function weight together with its type declaration which computes the weight of all Goods in a train of type Train Cargo. For the definition of weight, use only one defining equation where the right-hand side is a call to the function fold.

For example, weight aTrain should be evaluated to 300.

d) In this part of the exercise, you should create a program that controls a robber. The goal of the robber is to steal Goods, but when he tries to transport more than 1000 kg, he gets too slow and is caught by the police.

Implement a function robTrain :: Train Cargo -> IO (). Its input is a Train that just contains Goods. It ignores the Locomotive and processes the remainder of the Train as follows:

For a wagon with n kg Goods, it prints "Do you want to pick up the goods? (y|n)". If the user answers "y", it prints "You have stolen n kg goods.". Otherwise, there is no output.

Afterwards, if the accumulated Goods of the robber exceed the limit of 1000 kg, it prints "You were caught by the police." and terminates. Otherwise, if the whole Train has been processed (i.e., Empty is reached), it prints "You successfully robbed the train." and terminates. Otherwise, the program continues with the next part of the Train.

A successful run might look as follows:

*Main> robTrain (Locomotive (Wagon (Goods 1000) (Wagon (Goods 200) Empty)))
Do you want to pick up the goods? (y|n) n
Do you want to pick up the goods? (y|n) y
You have stolen 200 kg goods.
You successfully robbed the train.

In the following run, the user is caught by the police:
*Main> robTrain (Locomotive (Wagon (Goods 1000)) (Wagon (Goods 200) Empty)))
Do you want to pick up the goods? (y|n) y
You have stolen 1000 kg goods.
Do you want to pick up the goods? (y|n) y
You have stolen 200 kg goods.
You were caught by the police.

Hint: You should use the function `readLine :: IO String` to read the input from the user. To print a String, you should use the function `putStrLn :: String -> IO ()`.

Exercise 3 (Semantics): (10 + 10 + 6 = 26 points)

a) i) Let \( L_m \) = \{ [ ] \}, \{ [ [ ] ] \}, \ldots \}, i.e., \( L_m \) contains all lists where \( m \) opening brackets are followed by \( m \) closing brackets for an \( m \in \mathbb{N} \setminus \{0\} \). Let \( \leq_{nl} \subseteq \mathbb{N} \times \mathbb{N} \) be the relation that compares the nesting-level of two lists. More formally, if \( \text{nl}(x) \) is the nesting level of the list \( x \) and \( \leq \subseteq \mathbb{N} \times \mathbb{N} \) is the usual less-or-equal relation, then
\[ l \leq_{nl} l' \iff \text{nl}(l) \leq \text{nl}(l') \]

So we have, e.g., \([\square] \leq_{nl} [\square]\) because the nesting level of \([\square]\) is one and the nesting level of \([\square]\) is two.

1) Give an example for an infinite chain in \((L_\square, \leq_{nl})\).

2) Prove or disprove: the partial order \(\leq_{nl}\) is complete on \(L_\square\).

\[ l \leq_{len} l' \iff \text{len}(l) \leq \text{len}(l') \]

1) Give an example for an infinite chain in \((L_{\square}, \leq_{len})\).

2) Prove or disprove: the partial order \(\leq_{len}\) is complete on \(L_{\square}\).

b) i) Consider the following Haskell function \(f\):

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

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 for \(ff\). You may use full Haskell for \(ff\).

ii) Let \(\phi_{ff}\) be the semantics of the function \(ff\). Give the definition of \(\phi_{ff}^n(\bot)\) in closed form for any \(n \in \mathbb{N}\), i.e., give a non-recursive definition of the function that results from applying \(\phi_{ff}\) \(n\)-times to \(\bot\).

iii) Give the definition of the least fixpoint of \(\phi_{ff}\) in closed form.

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:

\[
\begin{array}{l}
\text{data Nats} = Z \mid S \text{ Nats} \\
\text{data Train a} = L (\text{Train a}) -- \text{a Locomotive} \\
\quad \mid W a (\text{Train a}) -- \text{a Wagon} \\
\quad \mid E -- \text{an Empty Train} \\
\text{data Cargo} = \text{NC} -- \text{No Cargo} \\
\quad \mid P \text{ Int} -- \text{n Persons}
\end{array}
\]

Give a graphical representation of the first three levels of the domain for the type \texttt{Train Cargo}. The third level contains the element \(\perp (P \perp) \perp\), for example. Note that the domain for the type \texttt{Train Cargo} also contains Trains with multiple locomotives, Trains without \(E\) at their ends, and so on. In other words, the assumption from Exercise 2 ("Assume that every Train consists of a single Locomotive at its beginning followed by a sequence of Wagons and Empty at its end.") does not hold for this exercise.
Solution:

a)  
   i)  1) \{[], [[]], [[[[]]], ...]}
   2) Consider the chain above. Since it contains infinitely many elements with increasing nesting level, its upper bounds have to have infinite nesting level. Since lists with infinite nesting level are not contained in \( L \), \( \leq_n \) is not a cpo.

   ii)  1) \{[], [0], [0, 0], ...\}
   2) The relation \( \leq_{ten} \) is a cpo iff \( L_0 \) has a least element w.r.t. \( \leq_{ten} \) and every \( \leq_{ten} \)-chain has a least upper bound in \( L_0 \). Obviously, the least element is the empty list \( [] \). Let \( C \) be a chain. If \( C \) is finite, then the longest list in \( C \) is the least upper bound. Otherwise, the infinite list \( l_\infty \) containing only zeros (as defined by \( \text{zeros}=0:zeros \)) is the least upper bound. Thus, \( \leq_{ten} \) is a cpo.

b)  
   i) \( \text{ff} :: ((\text{Int}, \text{Int}) \to \text{Int}) \to ((\text{Int}, \text{Int}) \to \text{Int}) \)
   \( \text{ff} \ f \ (x, 0) = 1 \)
   \( \text{ff} \ f \ (x, y) = x * f \ (x, y - 1) \)

   ii)
   \[
   (\phi^n_{tt}(\bot))(x, y) = \begin{cases} 
   1 & \text{if } y = 0 \land 0 < n \\
   x^y & \text{if } 0 < y < n \land x \neq \bot \\
   \bot & \text{otherwise} 
   \end{cases}
   
   \]

   iii)
   \[
   (\text{lfp} \phi_{tt})(x, y) = \begin{cases} 
   1 & \text{if } y = 0 \\
   x^y & \text{if } 0 < y \land x \neq \bot \\
   \bot & \text{otherwise} 
   \end{cases}
   
   \]

c)  
   \[
   L \ (L \ \bot) \quad L \ (W \ \bot) \quad L \ E \quad W \bot \ (L \ \bot) \quad W \bot \ (W \ \bot) \quad W \bot \ E \quad W \ NC \bot \quad W \ (P \ \bot) \ \bot
   
   \]

Exercise 4 (Lambda Calculus):

   (4 + 8 + 5 = 17 points)

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

   \[
   f' :: \text{Int} \to \text{Int} \to \text{Int}
   f' \ x \ 0 = 1
   f' \ x \ y = x * f' \ x \ (y - 1)
   \]
Please implement this function in the Lambda Calculus, i.e., give a term \( f \) such that, for all \( x, y, z \in \mathbb{Z} \),

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

b) Let

\[ t = \lambda \text{add} \ x \ y. (y == 0) \ x \ (\text{add} \ (x + 1) \ (y - 1)) \]

and

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

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

Please reduce \( \text{fix } t \ 0 \ 0 \) 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

\[ \lambda \text{add} \ x \ y. (y == 0) \ x \ (\text{add} \ (x + 1) \ (y - 1)) \]

whenever possible.

c) Consider the Boolean operator \text{nor} \ where \( \text{nor}(x, y) \) holds if and only if \( \text{or}(x, y) \) 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 \( \text{nor} \) in \( \rightarrow_{\beta}\)-normal form.

Solution:

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

b) 

\[ \text{fix} \ t \ 0 \ 0 \]

\[ \rightarrow_{\delta} (\lambda f. (f \ (\text{fix } f))) \ t \ 0 \ 0 \]

\[ \rightarrow_{\beta} t \ (\text{fix } t) \ 0 \ 0 \]

\[ \rightarrow_{\beta} (\lambda x \ y. (y == 0) \ x \ ((\text{fix } t) \ (x + 1) \ (y - 1))) \ 0 \ 0 \]

\[ \rightarrow_{\beta} (\lambda y. (y == 0) \ 0 \ ((\text{fix } t) \ (0 + 1) \ (y - 1))) \ 0 \]

\[ \rightarrow_{\beta} \text{if True } 0 \ ((\text{fix } t) \ (0 + 1) \ (0 - 1)) \]

\[ \rightarrow_{\delta} \text{if True } 0 \ ((\text{fix } t) \ (0 + 1) \ (0 - 1)) \]

\[ \rightarrow_{\beta} (\lambda x \ y. x) \ 0 \ ((\text{fix } t) \ (0 + 1) \ (0 - 1)) \]

\[ \rightarrow_{\beta} (\lambda y. 0) \ ((\text{fix } t) \ (0 + 1) \ (0 - 1)) \]

\[ \rightarrow_{\beta} 0 \]
c)

\[
\begin{align*}
nor &= \lambda x \ y \ (\lambda x \ y \ y) \ (y \ x \ (\lambda x \ y \ x)) \\
&\quad \text{or} \\
nor &= \lambda x \ y \ (\lambda x \ y \ y) \ (y \ (\lambda x \ y \ y) \ (\lambda x \ y \ x)) \\
&\quad \text{or} \\
nor &= \lambda x \ y \ (x \ y \ y) \ (\lambda x \ y \ y) \ (\lambda x \ y \ x) \\
&\quad \text{or}
\end{align*}
\]
...

Exercise 5 (Type Inference): (10 points)

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

Solution:

\[
W(A_0, \lambda f. f (f x)) \\
W(A_0 + \{f :: b_1\}, f (f x)) \\
W(A_0 + \{f :: b_1\}, f) = (id, b_1) \\
W(A_0 + \{f :: b_1\}, f x) \\
W(A_0 + \{f :: b_1\}, x) = (id, b_2) \\
mgu(b_1, b_2 \rightarrow b_3) = [b_1/b_2 \rightarrow b_3] \\
= [b_1/b_3, b_2/b_3] \\
mgu(b_2 \rightarrow b_3, b_1 \rightarrow b_4) = [b_2/b_3, b_4/b_3] \\
= [b_1/b_3, b_2/b_3, b_4/b_3, b_3/b_3] \\
= [b_1/b_3, b_2/b_3, b_4/b_3, (b_3 \rightarrow b_3) \rightarrow b_3]
\]