Exercise 1 \((4 + 5 + 4 + 5 = 18 \text{ points})\)

The following data structure represents polymorphic binary trees that contain values only in special `Value` nodes that have a single successor:

\[
\text{data } \text{Tree } a = \text{Leaf} \mid \text{Node} (\text{Tree } a) (\text{Tree } a) \mid \text{Value } a (\text{Tree } a)
\]

Consider the tree \(t\) of characters on the right-hand side. The representation of \(t\) as an object of type `Tree Char` in Haskell would be:

\[
(\text{Node} (\text{Value} 'a' (\text{Value} 'b' \text{Leaf})) (\text{Node} (\text{Node} \text{Leaf} \text{Leaf}) (\text{Value} 'c' \text{Leaf})))
\]

Implement the following functions in Haskell.

(a) The function `foldTree` of type

\[
(a \to b \to b) \to (b \to b \to b) \to b \to \text{Tree } a \to b
\]

works as follows: \(\text{foldTree } f \ g \ h \ x \) replaces all occurrences of the constructor `Value` in the tree \(x\) by \(f\), it replaces all occurrences of the constructor `Node` in \(x\) by \(g\), and it replaces all occurrences of the constructor `Leaf` in \(x\) by \(h\). So for the tree \(t\) above,

\[
\text{foldTree } (:) (++) \text{[]} t
\]

should compute

\[
((++) ((:) 'a' ((:) 'b' [])) ((++) ((++) [] [])) ((:) 'c' [])),
\]

which in the end results in "abc" (i.e., in the list \('[a', 'b', 'c']\)). Here, `Value` is replaced by \((:)\), `Node` is replaced by `([],)` and `Leaf` is replaced by `[]`.

\[
\text{foldTree } f \ g \ h \ (\text{Value } n \ x) = f \ n \ (\text{foldTree } f \ g \ h \ x)
\]

\[
\text{foldTree } f \ g \ h \ (\text{Node } x \ y) = g \ (\text{foldTree } f \ g \ h \ x) \ (\text{foldTree } f \ g \ h \ y)
\]

\[
\text{foldTree } \_ \_ \_ h \text{Leaf} = h
\]
(b) Use the \texttt{foldTree} function from (a) to implement the \texttt{average} function which has the type \texttt{Tree Int -> Int} and returns the average of the values that are stored in the tree. This should be accomplished as follows:

- Use \texttt{foldTree} with suitable functions as arguments in order to compute the \textit{sum} of the values stored in the trees.
- Use \texttt{foldTree} with suitable functions as arguments in order to compute the \textit{number of Value-objects in the tree}.
- Perform integer division with the pre-defined function \texttt{div :: Int -> Int -> Int} on these values to obtain the result.

Here your function is required to work correctly only on those trees that contain the constructor \texttt{Value} at least once.

\[
\text{average } t = \text{div } (\text{foldTree } (+) (+) 0 t) (\text{foldTree } (\lambda x y \rightarrow y+1) (+) 0 t)
\]
(c) Consider the following data type declaration for natural numbers:

\[
data \text{Nats} = \text{Zero} \mid \text{Succ} \text{Nats}
\]

A graphical representation of the first four levels of the domain for \text{Nats} could look like this:

\[
\begin{align*}
\text{Succ (Succ Zero)} & \quad \text{Succ (Succ (Succ \bot))} \\
\text{Succ Zero} & \quad \text{Succ (Succ \bot)} \\
\text{Zero} & \quad \text{Succ \bot} \\
\bot &
\end{align*}
\]

Sketch a graphical representation of the first three levels of the domain for the data type \text{Tree Bool}.

\[
\begin{align*}
\text{Value \bot (Node \bot \bot)} & \quad \text{Value \bot (Value \bot \bot)} & \quad \text{Node \bot (Value \bot \bot)} & \quad \text{Node \bot (Node \bot \bot)} \\
\text{Value \bot Leaf} & \quad \text{Value True \bot} & \quad \text{Value False \bot} & \\
\text{Node \bot Leaf} & \quad \text{Node Leaf \bot} & \quad \text{Node (Node \bot \bot) \bot} & \quad \text{Node (Value \bot \bot) \bot} \\
\text{Node \bot} & \quad \text{Leaf} & \quad \text{Node \bot} &
\end{align*}
\]
(d) Write a Haskell function `printStars` that first reads a string from the user, then prints this string on the console, converts the string to a number `n` (using the pre-defined function `read`) and in the end also prints `n` times the character ‘*’ on the console. Also give the type declaration for your function.

You may use the `do`-notation, but you are not obliged to use it. You do not have to check whether the input string is really a number. Some of the following pre-defined functions can be helpful:

- `getLine :: IO String` reads a string from the user
- `read :: String -> Int` converts a string to a number
- `putStrLn :: String -> IO ()` writes a string to the console

An example run should look as given below. Here the string “7” was read from the user.

```
Main> printStars
7
7*******
```

-- without do-notation
```
printStars :: IO ()
printStars = getLine >>= \s -> putStrLn s >> putStrLn (take (read s) (repeat '*'))
```

-- alternative: with do-notation
```
printStars2 :: IO ()
printStars2 = do s <- getLine
    putStrLn s
    putStrLn (take (read s) (repeat '*'))
```
Exercise 2 (4 + 5 = 9 points)

Consider the following Haskell declarations for the fib function, which for a natural number \(x\) computes the value fibonacci(\(x\)):

\[
\begin{align*}
\text{fib} & : \text{Int} \rightarrow \text{Int} \\
\text{fib} 0 & = 0 \\
\text{fib} 1 & = 1 \\
\text{fib} (x+2) & = \text{fib} (x+1) + \text{fib} x
\end{align*}
\]

(a) Please give the Haskell declarations for the higher-order function \(\text{f}_\text{fib}\) corresponding to \(\text{fib}\), i.e., the higher-order function \(\text{f}_\text{fib}\) such that the least fixpoint of \(\text{f}_\text{fib}\) is \(\text{fib}\). In addition to the function declaration(s), please also give the type declaration of \(\text{f}_\text{fib}\). Since you may use full Haskell for \(\text{f}_\text{fib}\), you do not need to translate \(\text{fib}\) into simple Haskell.

\[
\begin{align*}
\text{f}_\text{fib} & : (\text{Int} \rightarrow \text{Int}) \rightarrow (\text{Int} \rightarrow \text{Int}) \\
\text{f}_\text{fib} \text{fib} 0 & = 0 \\
\text{f}_\text{fib} \text{fib} 1 & = 1 \\
\text{f}_\text{fib} \text{fib} (x+2) & = \text{fib} (x+1) + \text{fib} x
\end{align*}
\]

(b) We add the Haskell declaration \(\text{bot} = \text{bot}\). For each \(n \in \mathbb{N}\) please determine which function is computed by \(\text{f}_\text{fib}^n \text{bot}\). Here “\(\text{f}_\text{fib}^n \text{bot}\)” represents the \(n\)-fold application of \(\text{f}_\text{fib}\) to \(\text{bot}\), i.e., it is short for \(\underbrace{\text{f}_\text{fib} (\text{f}_\text{fib} \ldots (\text{f}_\text{fib} \text{bot})\ldots)}_{n \text{ times}}\).

Let \(f_n : \mathbb{Z}_\bot \rightarrow \mathbb{Z}_\bot\) be the function that is computed by \(\text{f}_\text{fib}^n \text{bot}\).

Give \(f_n\) in closed form, i.e., using a non-recursive definition. In this definition, you may use the function \(\text{fibonacci} : \mathbb{N} \rightarrow \mathbb{N}\) where \(\text{fibonacci}(x)\) computes the \(x\)-th Fibonacci number. Here it suffices to give the result of your calculations. You do not need to present any intermediate steps.

\[
(f_{\text{fib}}^n(\bot))(x) = \begin{cases} 
\text{fibonacci}(x), & \text{if } n > 0 \text{ and } 0 \leq x \leq n \\
\bot, & \text{otherwise}
\end{cases}
\]
Exercise 3 (3 + 3 = 6 points)

Let $D_1, D_2, D_3$ be domains with corresponding complete partial orders $\sqsubseteq_{D_1}, \sqsubseteq_{D_2}, \sqsubseteq_{D_3}$. As we know from the lecture, then also $\sqsubseteq_{(D_2 \times D_3)_{\perp}}$ is a complete partial order on $(D_2 \times D_3)_{\perp}$.

Now let $f : D_1 \to D_2$ and $g : D_1 \to D_3$ be functions.
We then define the function $h : D_1 \to (D_2 \times D_3)_{\perp}$ via $h(x) = (f(x), g(x))$.

(a) Prove or disprove: If $f$ and $g$ are strict functions, then also $h$ is a strict function.

The statement does not hold. Consider the following counterexample: $D_1 = D_2 = D_3 = \mathbb{B}_{\perp}$ and $f = g = \bot_{\mathbb{B}_{\perp} \to \mathbb{B}_{\perp}}$. Obviously $f$ and $g$ are strict functions, i.e., $f(\bot_{\mathbb{B}_{\perp}}) = g(\bot_{\mathbb{B}_{\perp}}) = \bot_{\mathbb{B}_{\perp}}$. However, we have $h(\bot_{\mathbb{B}_{\perp}}) = (\bot_{\mathbb{B}_{\perp}}, \bot_{\mathbb{B}_{\perp}}) \neq \bot_{(\mathbb{B}_{\perp} \times \mathbb{B}_{\perp})_{\perp}}$.

(b) Prove or disprove: If $f$ and $g$ are monotonic functions, then also $h$ is a monotonic function.

Let $x \sqsubseteq_{D_1} y$. Then we have:

\[
\begin{align*}
\sqsubseteq_{(D_2 \times D_3)_{\perp}} & \quad (f(y), g(y)) \\
\sqsubseteq_{(D_2 \times D_3)_{\perp}} & \quad (f(x), g(x)) \\
\end{align*}
\]

$f$ and $g$ are monotonic, def. of $\sqsubseteq_{(D_2 \times D_3)_{\perp}}$.

Hence, also $h$ is monotonic. \qed
Exercise 4 (6 points)

We define the following data structures for natural numbers and polymorphic lists:

```haskell
data Nats = Zero | Succ Nats
data List a = Nil | Cons a (List a)
```

Consider the following expression in complex Haskell:

```haskell
let get n Nil = Zero
    get Zero (Cons x xs) = x
    get (Succ n) (Cons x xs) = get n xs
in get
```

Please give an equivalent expression `let get = ... in get` in simple Haskell.

Your solution should use the functions defined in the transformation from the lecture such as `sel_{n,i}`, `isa_{constr}`, and `argof_{constr}`. However, you do not have to use the transformation rules from the lecture.

```haskell
let get = \n -> \xs -> if (isa_{Nil} xs)
    then Zero
    else if (isa_{Zero} n)
        then (sel_{2,1} (argof_{Cons} xs))
        else get (sel_{1,1} (argof_{Succ} n)) (sel_{2,2} (argof_{Cons} xs))
    in get
```
Exercise 5 (4 + 5 = 9 points)

Consider the following data structure for polymorphic lists:

```haskell
data List a = Nil | Cons a (List a)
```

(a) Please translate the following Haskell expression into an equivalent lambda term (e.g., using \textit{Lam}). Recall that pre-defined functions like \textit{odd} or \textit{(+)} are translated into constants of the lambda calculus.

It suffices to give the result of the transformation.

```haskell
let f = \x -> if (odd x) then (\y -> x) else f ((+) x 3)
in f
```

```latex
\text{fix} \ (\lambda f. \text{if} \ (\text{odd} \ x) \ (\lambda y. x) \ \text{else} \ f \ ((+) \ x \ 3) )
```
(b) Let \( \delta \) be the set of rules for evaluating the lambda terms resulting from Haskell, i.e., \( \delta \) contains at least the following rules:

\[
\begin{align*}
\text{fix} & \rightarrow \lambda f. f (\text{fix } f) \\
\text{times 3 2} & \rightarrow 6
\end{align*}
\]

Now let the lambda term \( t \) be defined as follows:

\[
t = (\lambda x. (\text{fix } \lambda g. x)) \ (\lambda z. (\text{times 3 2}))
\]

Please reduce the lambda term \( t \) by WHNO-reduction with the \( \rightarrow_{\beta\delta} \)-relation. You have to give all intermediate steps until you reach \textit{weak head normal form} (and no further steps).

\[
\begin{align*}
(\lambda x. (\text{fix } \lambda g. x)) \ (\lambda z. (\text{times 3 2})) & \rightarrow_{\beta} \text{fix } (\lambda g. \lambda z. (\text{times 3 2})) \\
& \rightarrow_{\delta} (\lambda f. f (\text{fix } f)) \ (\lambda g. \lambda z. (\text{times 3 2})) \\
& \rightarrow_{\beta} (\lambda g. \lambda z. (\text{times 3 2})) \ (\text{fix } (\lambda g. \lambda z. (\text{times 3 2}))) \\
& \rightarrow_{\beta} \lambda z. (\text{times 3 2})
\end{align*}
\]
Exercise 6 (10 points)

Use the type inference algorithm $W$ to determine the most general type of the following lambda term under the initial type assumption $A_0$. Show the results of all sub-computations and unifications, too. If the term is not well typed, show how and why the $W$-algorithm detects this.

$$((\text{Cons } \lambda x. x) \; y)$$

The initial type assumption $A_0$ contains at least the following:

$$A_0(\text{Cons}) = \forall a. (a \to (\text{List } a \to \text{List } a))$$
$$A_0(x) = \forall a. a$$
$$A_0(y) = \forall a. a$$

$W(A_0, ((\text{Cons } \lambda x. x) \; y))$
$W(A_0, (\text{Cons } \lambda x. x))$
$W(A_0, \text{Cons})$
\begin{align*}
&= (\text{id}, (b_1 \to (\text{List } b_1 \to \text{List } b_1))) \\
W(A_0, \lambda x. x)
&= (\text{id}, b_2) \\
W(A_0 + \{x :: b_2\}, x)
&= (\text{id}, b_2) \\
mgu( b_1 \to (\text{List } b_1 \to \text{List } b_1), ((b_2 \to b_2) \to b_3) )
&= [ b_1/(b_2 \to b_2), b_3/(\text{List } b_2 \to b_2) \to \text{List } (b_2 \to b_2)] \\
W(A_0, y)
&= (\text{id}, b_4) \\
mgu( (\text{List } (b_2 \to b_2) \to \text{List } (b_2 \to b_2)), (b_4 \to b_5) )
&= [ b_4/\text{List } (b_2 \to b_2), b_5/\text{List } (b_2 \to b_2)] \\
&= ( b_1/(b_2 \to b_2), b_3/(\text{List } (b_2 \to b_2) \to \text{List } (b_2 \to b_2)), b_4/\text{List } (b_2 \to b_2), b_5/\text{List } (b_2 \to b_2), \text{List } (b_2 \to b_2) )
\end{align*}

Resulting type: $\text{List } (b_2 \to b_2)$