Exercise 1 (Programming in Haskell): (5 + 12 + 9 + 14 = 40 points)

We use the following data structure `Tree a` to represent binary trees in Haskell.

```
data Tree a = E | N a (Tree a) (Tree a)
```

For example, the tree from Fig. 1 can be represented by:

```
exTree :: Tree Int
exTree = N 16 (N 37 (N 19 E E) (N 21 E E)) (N 25 (N 2 E E) (N 12 E E))
```

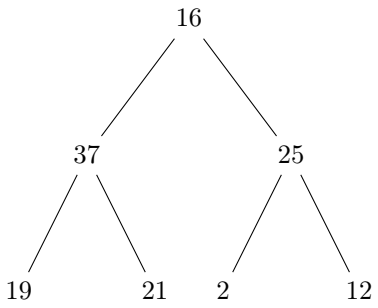


Figure 1: Tree of type `Tree Int`

In each of the following exercises you can use functions from the Haskell `Prelude` and from previous exercises even if you did not implement them. Moreover, you can always implement auxiliary functions.

a) Declare `Tree a` as an instance of the type class `Ord` whenever `a` is an instance of `Ord`. In this instance declaration, you should provide an implementation of the function `<=` to compare two trees. For two non-empty trees `t1` and `t2`, we have `t1 <= t2` if and only if the value of `t1`'s root node is smaller or equal to the value of `t2`'s root node. For example, `(exTree <= N 19 E E) == True` but `(N 19 E E <= exTree) == False`, as `(19 <= 16) == False`. Furthermore, `(t <= E) == True` for any `t::Tree a` but `(E <= t) == True` if and only if `t == E`.

Hints:

- You can assume that `Tree a` is an instance of `Eq` whenever `a` is an instance of `Eq`.

b) A *heap* is a binary tree in which a value stored at a node is greater or equal to all values stored in its children. For example, the tree in Fig. 1 is not a heap, as `(16 >= 37) == False`. However, the subtree of the tree in Fig. 1 with root 25 is a heap, as `(25 >= 2) == True` and `(25 >= 12) == True`. Note that the root node of a heap contains the *maximal* value stored in the heap. The empty tree `E` is also a heap.

Write a Haskell–function `heapify` and also give its type declaration. Its only argument is an element `t::Tree a` for a type `a` of the type class `Ord`. The call `heapify t` for some `t::Tree a` results in a tree `h::Tree a` which stores exactly the same values as `t` but is also a heap. If a value is stored n times in `t`, then it is also stored n times in `heapify t`.

For a tree `N v l r`, your implementation should check if `v` is greater or equal to the values stored in `l` and `r`, respectively. If yes, the result is `N v lh rh`, where `lh` is `l` transformed into a heap and `rh` is `r` transformed into a heap. If no, then swap `v` and the maximal value of the tree. Afterwards, call heapify on the subtree where `v` was swapped to in order to ensure that the result is a heap.

For example, `heapify exTree` results in the tree in Fig. 2.

c) For the following exercise, assume that there is a function `deleteRoot::Ord a => Tree a -> Tree a`. For a given heap `h` with `h /= E`, the call `deleteRoot h` results in a heap storing exactly the same values as `h` *without* its root node. Again, if a value is stored n times in `h` without its root node, then the value is also stored n times in `deleteRoot h`. Furthermore, assume that there is a function `insert::a -> Tree a -> Tree a` that inserts a value into a tree.

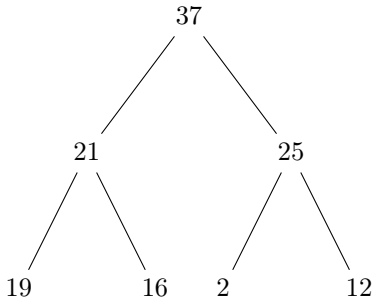


Figure 2: Heapified version of Fig. 1

Write a Haskell–function `heapsort` and also give its type declaration. Its only argument is a list `xs::[a]` for a type `a` of the type class `Ord`. The function `heapsort` sorts `xs` in *decreasing* order by first transforming it into a heap. Then, it extracts the maximal value of the heap by using `deleteRoot` and adds this value to the resulting list. This is repeated until the heap is empty.

Hints:

- You can use the functions `heapify`, `insert`, and `foldr` to transform a list into a heap.

d) Implement a function `participants::[(String, String)] -> IO ()`. This function simulates an interactive system for managing participants encoded as pairs `(surname, first_name)`.

When `participants xs` is called, it asks the user for an input. If the input is

- `"get"`, then the user can enter a value `k::Int` and gets the `k`th name when sorting the participants w.r.t. surnames in decreasing order. Participants with the same surnames are sorted by their first names. The first participant has the index 0. If there are less than `k+1` participants, an error message is prompted. Then `participants` calls itself again with the same argument.
- `"add"`, then the user enters first the surname and then the first name of the participant to be added. Then `participants` prompts a success message and calls itself again with the list resulting from `xs` when adding the entered participant.
- In any other case the program terminates.

An example run should look as follows. Here, inputs of the user are written in *italics*.

```
*Main> participants []
Add participant or get kth?
```
add
Giesl
Juergen
```
Added participant
Add participant or get kth?
```
add
Mueller
Julia
```
Added participant
Add participant or get kth?
```
get
2
```
Not enough participants
Add participant or get kth?
```

get

1

```
Giesl, Juergen
Add participant or get kth?
```

Stop

```
*Main>
```

Hints:

- The function `putStrLn::String -> IO()` prints a string and performs a line break.
- The function `getLine::IO String` reads a string from the keyboard.
- You can assume that there is a function `getInt::IO Int` that reads an integer from the keyboard.
- You can use the indexing operator `(!!)::[a] -> Int -> a`. Here, `xs!!k` gives the `k`th element of the list `xs` whenever `k` is between `0` and `(length xs)-1`.
- You can use the function `heapsort` to sort the list of participants in decreasing order. The reason is that for two pairs `(s1,s2),(t1,t2)::(String,String)` we have `(s1,s2) <= (t1,t2)` iff `s1 <= t1` or `s1 == t1` and `s2 <= t2`. Strings are compared as usual by `<=`.

Solution:

```
data Tree a = E | N a (Tree a) (Tree a) deriving Show

instance Eq a => Eq (Tree a) where
  (==) E E = True
  (==) (N v1 l1 r1) (N v2 l2 r2) = v1 == v2
  (==) _ _ = False

--Help
insert :: Ord a => a -> Tree a -> Tree a
insert x E = N x E E
insert x (N v l r) | x < v = N v (insert x l) r
                   | otherwise = N v l (insert x r)

deleteLeaf :: Tree a -> (a, Tree a)
deleteLeaf (N v E E) = (v, E)
deleteLeaf (N v (N v1 l1 r1) r) = let (v3, l3) = deleteLeaf (N v1 l1 r1) in (v3, (N v l3 r))
deleteLeaf (N v E (N v2 l2 r2)) = let (v4, r4) = deleteLeaf (N v2 l2 r2) in (v4, (N v E r4))

deleteRoot :: Ord a => Tree a -> Tree a
deleteRoot (N v E E) = E
deleteRoot (N v l r) = let (x, N w l1 r1) = deleteLeaf (N v l r) in (heapify (N x l1 r1))

getInt :: IO Int
getInt = do
  x <- getLine
  return (read x :: Int)
```

a)
```
instance Ord a => Ord (Tree a) where
     (<=) _ E = True
     (<=) (N v1 l1 r1) (N v2 l2 r2) = v1 <= v2
     (<=) _ _ = False
```

b)
```
heapify :: Ord a => Tree a -> Tree a
heapify E = E
heapify (N v E E) = N v E E
heapify (N v l E) = let (N v1 l1 r1) = lh in
                        if (v >= v1) then (N v lh E) else N v1 (heapify (N v l1 r1)) E
                    where lh = (heapify l)
heapify (N v E r) = let (N v2 l2 r2) = rh in
                        if (v >= v2) then (N v E rh) else N v2 E (heapify (N v l2 r2))
                    where rh = (heapify r)
heapify (N v l r) | ((N v l r) >= lh) && ((N v l r) >= rh) = N v lh rh
                  | lh >= rh = let (N v1 l1 r1) = lh in N v1 (heapify (N v l1 r1)) rh
                  | lh < rh = let (N v2 l2 r2) = rh in N v2 lh (heapify (N v l2 r2))
                    where
                     (lh, rh) = ((heapify l), (heapify r))
```

c)
```
heapsort :: Ord a => [a] -> [a]
heapsort xs = toList (heapify (foldr insert E xs)) where
  toList E = []
  toList (N v l r) = v:toList(deleteRoot (N v l r))
```

d)
```
participants :: [(String, String)] -> IO ()
participants xs = do
  putStrLn "Add participant or get kth?"
  x <- getLine
  if (x == "get")
  then do
    k <- getInt
    if (k < 0 || k >= (length xs))
        then do
          putStrLn "Not enough participants"
          participants xs
        else
        let (sur, first) = (heapsort xs) !! k in do
                                              putStrLn (sur ++ ", " ++ first)
                                              participants xs
  else if (x == "add") then
        do
          sur <- getLine
          first <- getLine
          putStrLn "Added participant"
          participants ((sur, first) : xs)
       else return ()
```

Exercise 2 (Semantics): ((10 + 7) + (5 + 7 + 4) + 7 = 40 points)

a) i) Prove the following generalization of the Fixpoint Theorem. Let D be a domain, $\sqsubseteq$ a complete partial order on D, $f : D \to D$ a continuous function w.r.t. $\sqsubseteq$, and $d \in D$. If $d \sqsubseteq f(d)$ then $p^* = \sqcup\{f^n(d) \mid n \in \mathbb{N}\}$ satisfies the following:

- The element p^* is a fixpoint of f.
- For every fixpoint p of f with $d \sqsubseteq p$ we have $p^* \sqsubseteq p$.

Hints:

- You do *not* have to show that $\sqcup\{f^n(d) \mid n \in \mathbb{N}\}$ exists, i.e., that $\{f^n(d) \mid n \in \mathbb{N}\}$ is indeed a chain.

ii) Let D be a domain, $\sqsubseteq$ a complete partial order on D, and $f : D \to D$ a continuous function w.r.t. $\sqsubseteq$. By the Fixpoint Theorem, f has a least fixpoint lfp f. Disprove the following claim by giving a *counterexample*:

$$\text{If } d \in D \text{ with } d \sqsubseteq f(d) \text{ then } d \sqsubseteq \text{lfp } f.$$

Hints:

- You also have to show that your counterexample meets the requirements, i.e., that the chosen order is complete, that your function f is continuous, and that your chosen $d \in D$ satisfies $d \sqsubseteq f(d)$.
- You may use that any *flat* domain with the order $x \sqsubseteq y$ iff $x = y \vee x = \bot$ is complete.
- On *flat* domains, continuity and monotonicity are equivalent.

b) i) Consider the following Haskell function `f`:

```
f :: (Int, Int) -> Int
f (n, 0) = 1
f (0, k) = 0
f (n, k) = f(n-1, k) + f(n-1, k-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 for `ff`. You may use full Haskell for `ff`.

ii) Let $\phi_{\texttt{ff}}$ be the semantics of the function `ff`. Give the definition of $\phi_{\texttt{ff}}^m(\bot)$ in closed form for any $m \in \mathbb{N}$, i.e., give a non-recursive definition of the function that results from applying $\phi_{\texttt{ff}}$ m-times to $\bot$. Here, you should assume that `Int` can represent all integers, so no overflow can occur.

Hints:

- The binomial coefficient $\binom{n}{k}$ for $n \in \mathbb{N}$, $k \in \mathbb{Z}$ is defined as follows:

$$\binom{n}{k} = \begin{cases} \frac{n!}{k! \cdot (n-k)!}, & 0 \leq k \leq n \\ 0, & k < 0 \text{ or } k > n \end{cases}.$$

- For $n \in \mathbb{N}$, $k \in \mathbb{Z}$ we have $\binom{n+1}{k} = \binom{n}{k} + \binom{n}{k-1}$.

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

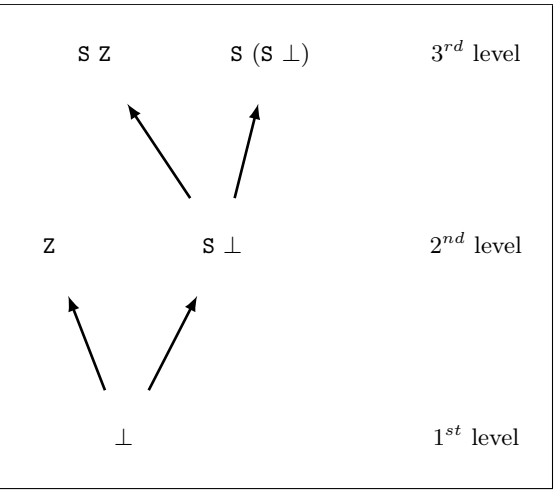
c) Consider the data type declarations on the left and, as an example, the graphical representation of the first three levels of the domain for `Nats` on the right:

```
data Nats = Z | S Nats

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

data Tree a = E | N a (Tree a) (Tree a)

data Unit = U ()
```



Give a graphical representation of the first three levels of the domain for the type `Tree Unit`.

Solution:

a) i) 1. p^* is a fixpoint of f:
We have

$$\begin{aligned}
& f(p^*) \\
=& f(\sqcup\{f^n(d) \mid n \in \mathbb{N}\}) \\
=& \sqcup\{f(f^n(d)) \mid n \in \mathbb{N}\} && \text{(Continuity)} \\
=& \sqcup\{f^{n+1}(d) \mid n \in \mathbb{N}\} \\
=& \sqcup\left(\{f^{n+1}(d) \mid n \in \mathbb{N}\} \cup \{d\}\right) && (d \sqsubseteq f(d)) \\
=& \sqcup\{f^n(d) \mid n \in \mathbb{N}\} \\
=& p^*.
\end{aligned}$$

So, $\sqcup\{f^n(d) \mid n \in \mathbb{N}\}$ is a fixpoint of f.

2. $p^* \sqsubseteq p$ for all fixpoints p with $p \sqsubseteq d$:
   Take any fixpoint p, such that $d \sqsubseteq p$. We will show that for every $n \in \mathbb{N}$ we have $f^n(d) \sqsubseteq p$ by using induction on n.
   (I.B.) $f^0(d) = d \sqsubseteq p$ by assumption.
   (I.H.) Assume that for a fixed $n \in \mathbb{N}$ we have $f^n(d) \sqsubseteq p$.
   (I.S.)

$$\begin{aligned} f^{n+1}(d) &= f(f^n(d)) \\ &\sqsubseteq f(p) && \text{(I.H. and continuity } \Rightarrow \text{ monotonicity)} \\ &= p. && \text{(as } p \text{ is a fixpoint of } f\text{)} \end{aligned}$$

   But this means that p is an upper bound of $\{f^n(d) \mid n \in \mathbb{N}\}$, so we must have $p^* = \sqcup\{f^n(d) \mid n \in \mathbb{N}\} \sqsubseteq p$ as p^* is the least upper bound. This concludes the proof.

ii) Take $\mathbb{Z}_\perp$ with the order $x \sqsubseteq y$ iff $x = y \vee x = \perp$. By the hint, this is a complete order. The identity function $id_{\mathbb{Z}_\perp} : \mathbb{Z}_\perp \to \mathbb{Z}_\perp, x \mapsto x$ is continuous as it is monotonic and $\mathbb{Z}_\perp$ is flat:

$$x \sqsubseteq y \Rightarrow id_{\mathbb{Z}_\perp}(x) = x \sqsubseteq y = id_{\mathbb{Z}_\perp}(y).$$

Alternatively, for any chain $S \subseteq \mathbb{Z}_\perp$ we have $id_{\mathbb{Z}_\perp}(S) = S$, i.e., $id_{\mathbb{Z}_\perp}(\sqcup S) = \sqcup S = \sqcup id_{\mathbb{Z}_\perp}(S)$.
Furthermore, $2 \sqsubseteq 2 = id_{\mathbb{Z}_\perp}(2)$. However, $id_{\mathbb{Z}_\perp}(\perp) = \perp$. So, the least fixpoint is $\perp$ but $2 \not\sqsubseteq \perp$.

b) i)

```
ff :: ((Int, Int) -> Int) -> ((Int, Int) -> Int)
ff g (n, 0) = 1
ff g (0, k) = 0
ff g (n, k) = g(n-1, k) + g(n-1, k-1)
```

ii) $\phi^0_{\texttt{ff}}(\perp) = \perp$

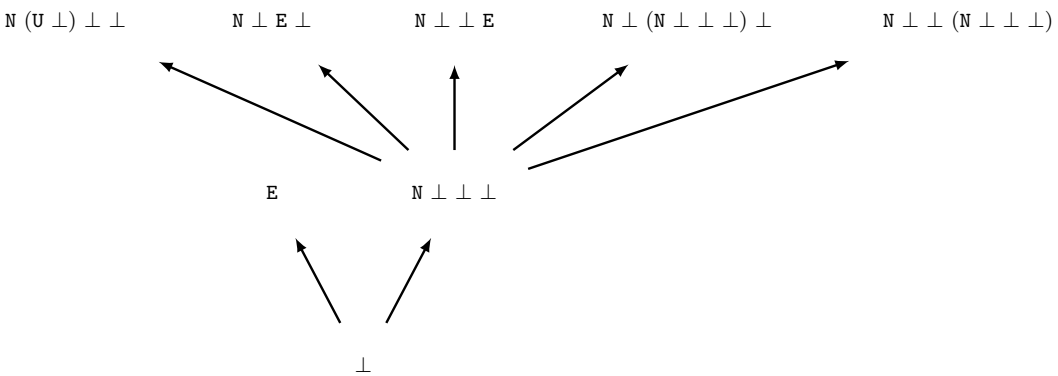
$$\phi^1_{\texttt{ff}}(\perp)(n,k) = \begin{cases} 1, & k = 0 \\ 0, & n = 0 \wedge k \notin \{0, \perp\} \\ \perp, & \texttt{otherwise} \end{cases} = \begin{cases} 1, & k = 0 \wedge n \notin \mathbb{N} \\ \binom{n}{k}, & 0 \leq n < 1 \wedge k \neq \perp \\ \perp, & \texttt{otherwise} \end{cases}$$

$$\phi^2_{\texttt{ff}}(\perp)(n,k) = \begin{cases} 1, & k = 0 \\ 0, & n = 0 \wedge k \notin \{0, \perp\} \\ 1, & n = k = 1 \\ 0, & n = 1 \wedge k \notin \{0, 1, \perp\} \\ \perp, & \texttt{otherwise} \end{cases} = \begin{cases} 1, & k = 0 \wedge n \notin \mathbb{N} \\ \binom{n}{k}, & 0 \leq n < 2 \wedge k \neq \perp \\ \perp, & \texttt{otherwise} \end{cases}$$

$$\phi^m_{\texttt{ff}}(\perp)(n,k) = \begin{cases} 1, & k = 0 \wedge n \notin \mathbb{N} \wedge m > 0 \\ \binom{n}{k}, & 0 \leq n < m \wedge k \neq \perp \\ \perp, & \texttt{otherwise} \end{cases}$$

iii) $\text{lfp}\ \phi_{\texttt{ff}}(n,k) = \begin{cases} 1, & k = 0 \wedge n \notin \mathbb{N} \\ \binom{n}{k}, & 0 \leq n \wedge k \neq \perp \\ \perp, & \texttt{otherwise} \end{cases}$

c)



Exercise 3 (Lambda Calculus): ((4 + 4) + 8 + 6 = 22 points)

a) Consider the following function that determines whether the first list is shorter or as long as than the second. Here, lists are represented by the data structure `List a` defined by `data List a = N | C a (List a)`.

```
cp :: List a -> List a -> Bool
cp N ys = True
cp (C x xs) N = False
cp (C x xs) (C y ys) = cp xs ys
```

i) Please give an equivalent function in *simple* Haskell.

ii) Implement the function `cp` in the *lambda calculus*, i.e., give a lambda term q such that, for all lists l_1, l_2 the term $q\ l_1\ l_2$ can be reduced to `True` if and only if l_1 is at most as long as l_2, and to `False` otherwise.

Hints:

- You do *not* have to use the transformation algorithms presented in the lecture. It is sufficient to just give an equivalent simple program and an equivalent lambda term.
- You can use infix notation for predefined functions like `(&&)` in both simple Haskell and the lambda calculus.

b) Let

$$t = \lambda\ g\ n.\ \texttt{if}\ (n == 0)\ (\lambda z.\texttt{True})\ (\lambda z.g\ 0)$$

and

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

Please reduce `fix` t 1 by WHNO-reduction with the $\to_{\beta\delta}$-relation. List **all** intermediate steps until reaching weak head normal form, but please write "t" instead of

$$\lambda\ g\ n.\ \texttt{if}\ (n == 0)\ (\lambda z.\texttt{True})\ (\lambda z.g\ 0)$$

whenever possible.

c) Consider the representation of natural numbers in the pure lambda calculus presented in the lecture (i.e., $n \in \mathbb{N}$ is represented by the term $\overline{n} = \lambda f\ x.\ f^n\ x$). Give a pure lambda term for the function `minus`, such that for $m, n \in \mathbb{N}$ the expression `minus` $\overline{m}\ \overline{n}$ can be reduced to $\begin{cases}\overline{m-n}, & \text{if } n \leq m\\ \overline{0}, & \texttt{otherwise}.\end{cases}$

Explain your solution shortly. You may give a reduction sequence as explanation.

Hints:

- You can assume that there is a pure lambda term `pred` such that `pred` $\overline{n}$ can be reduced to $\begin{cases}\overline{n-1}, & \text{if } n > 0\\ \overline{0}, & \text{if } n = 0.\end{cases}$

Solution:

a) i)

```
cp = \xs -> \ys ->
          if (isa_N xs) then True
            else if (isa_C xs)  && (isa_N ys) then False
              else if (isa_C xs)  && (isa_C ys)
                    then cp (sel_2,2 (argof_C xs)) (sel_2,2 (argof_C ys))
                   else bot
```

ii)

$$\begin{aligned}&\texttt{fix}\ (\lambda\ g\ xs\ ys.\texttt{if}\ (\texttt{isa\_N}\ xs)\ \texttt{True}\ (\texttt{if}\ (\texttt{isa\_C}\ xs\ \&\&\ \texttt{isa\_N}\ ys)\ \texttt{False}\\ &\quad (\texttt{if}\ (\texttt{isa\_C}\ xs\ \&\&\ \texttt{isa\_C}\ ys)\ (g\ (\texttt{sel\_2,2}\ (\texttt{argof\_C}\ xs))\ (\texttt{sel\_2,2}\ (\texttt{argof\_C}\ xs)))\ \texttt{bot})))\end{aligned}$$

b)

$$
\begin{aligned}
& \texttt{fix}\ t\ 1 \\
\rightarrow_\delta\ & (\lambda f.(f\ (\texttt{fix}\ f)))\ t\ 1 \\
\rightarrow_\beta\ & t\ (\texttt{fix}\ t)\ 1 \\
\rightarrow_\beta\ & (\lambda\ n.\ \texttt{if}\ (n == 0)\ (\lambda z.\texttt{True})\ (\lambda z.\texttt{fix}\ t\ 0))\ 1 \\
\rightarrow_\beta\ & \texttt{if}\ (1 == 0)\ (\lambda z.\texttt{True})\ (\lambda z.\texttt{fix}\ t\ 0) \\
\rightarrow_\delta\ & \texttt{if False}\ (\lambda z.\texttt{True})\ (\lambda z.\texttt{fix}\ t\ 0) \\
\rightarrow_\delta\ & (\lambda\ x\ y.y)\ (\lambda z.\texttt{True})\ (\lambda z.\texttt{fix}\ t\ 0) \\
\rightarrow_\beta\ & (\lambda\ y.y)\ (\lambda z.\texttt{fix}\ t\ 0) \\
\rightarrow_\beta\ & \lambda z.\texttt{fix}\ t\ 0
\end{aligned}
$$

c)

$$\texttt{minus} = \lambda\ m\ n.n\ \texttt{pred}\ m$$

The representation of n applies a function g n–times to some x. If $g = \texttt{pred}$ then this results in subtracting 1 exactly n times from m and in 0 if $m \not\geq n$.

$$
\begin{aligned}
& \texttt{minus}\ \overline{m}\ \overline{n} \\
\rightarrow_\beta\ & (\lambda\ n.n\ \texttt{pred}\ \overline{m})\ \overline{n} \\
\rightarrow_\beta\ & \overline{n}\ \texttt{pred}\ \overline{m} \\
\rightarrow_\beta\ & (\lambda x.\texttt{pred}^n\ x)\ \overline{m} \\
\rightarrow_\beta\ & \texttt{pred}^n\ \overline{m} \\
\rightarrow_\beta^*\ & \begin{cases} \overline{m-n}, & \text{if } n \leq m \\ \overline{0}, & \texttt{otherwise} \end{cases}
\end{aligned}
$$

Exercise 4 (Type Inference): (18 points)

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

Hints:

- When writing $\mathcal{W}(A, t) = (\theta, t)$, you do not have to give the full substitution θ, but it is enough to give the parts of θ that concern the free variables in the type schemas of A.

Solution:

$\mathcal{W}(A_0, \lambda x.g\ (f\ x)\ x)$
$\quad \mathcal{W}(A_0 + \{x :: b_1\}, g\ (f\ x)\ x)$

$$
\begin{array}{l}
\mathcal{W}(A_0 + \{x :: b_1\}, g\ (f\ x)) \\
\quad \mathcal{W}(A_0 + \{x :: b_1\}, g) \\
\quad = (id, \texttt{Bool} \to b_2 \to b_2) \\
\quad \mathcal{W}(A_0 + \{x :: b_1\}, f\ x) \\
\qquad \mathcal{W}(A_0 + \{x :: b_1\}, f) \\
\qquad = (id, b_3 \to b_3) \\
\qquad \mathcal{W}(A_0 + \{x :: b_1\}, x) \\
\qquad = (id, b_1) \\
\qquad \texttt{mgu}\,(b_3 \to b_3, b_1 \to b_4) = [b_3/b_1, b_4/b_1] \\
\quad = (id, b_1) \\
\quad \texttt{mgu}\,(\texttt{Bool} \to b_2 \to b_2, b_1 \to b_5) = [b_1/\texttt{Bool}, b_5/(b_2 \to b_2)] \\
= ([b_1/\texttt{Bool}], b_2 \to b_2) \\
\mathcal{W}(A_0 + \{x :: \texttt{Bool}\}, x) \\
= (id, \texttt{Bool}) \\
\texttt{mgu}\,(b_2 \to b_2, \texttt{Bool} \to b_6) = [b_2/\texttt{Bool}, b_6/\texttt{Bool}] \\
= ([b_1/\texttt{Bool}], \texttt{Bool}) \\
= (id, \texttt{Bool} \to \texttt{Bool})
\end{array}
$$

Under the type assumption A_0 the most general type of $\lambda x.g\ (f\ x)\ x$ is $\texttt{Bool} \to \texttt{Bool}$.