Exercise 1 (Quiz): (4 + 4 + 4 + 4 + 4 = 20 points)

Give a short proof sketch or a counterexample for each of the following statements:

a) Is $\sqsubseteq$ always a complete partial order for flat domains like $\mathbb{Z}_\perp, \mathbb{B}_\perp, \ldots$?

b) Can the function $f : \mathbb{Z}_\perp \to \mathbb{Z}$ with $f(x) = \begin{cases} 1 & \text{if } x \in \mathbb{Z} \text{ and } x \leq 0 \\ 0 & \text{otherwise} \end{cases}$ be implemented in Haskell?

c) Is $g : (\mathbb{Z} \to \mathbb{Z}_\perp) \to \mathbb{Z}_\perp$ with $g(h) = \begin{cases} 0 & \text{if } h(x) \neq \perp \quad \text{for all } x \in \mathbb{Z} \\ \perp & \text{otherwise} \end{cases}$ continuous?

d) If a lambda term t can be reduced to s with $\to_{\beta\delta}$ using an outermost strategy, can t also be reduced to s with $\to_{\beta\delta}$ using an innermost strategy? Here, you may choose an arbitrary delta-rule set δ.

e) The $\to_{\beta\delta}$ reduction in lambda calculus is confluent. Is Simple Haskell also confluent?

Solution:

a) **Yes**, because flat domains only have chains of finite length and a minimal element. Hence, by Theorem 2.1.13(a), $\sqsubseteq$ is a complete partial order.

b) **No**, as f is not continuous and thus, not computable: Consider the chain $S = \{\perp, 0\}$. There exists no least upper bound for $f(S) = \{0, 1\}$, and hence $f(\sqcup S) = \sqcup f(S)$ does not hold.

c) **No**. For a counterexample, let $f_i(x) = \begin{cases} 0 & \text{if } x \leq i \\ \perp & \text{otherwise} \end{cases}$.

Then for the chain $S = \{f_1, f_2, \ldots\}$, we have $\sqcup S = f_\infty$ with $f_\infty(x) = 0$ for all $x \in \mathbb{Z}$. Then $\sqcup g(S) = \perp \neq 0 = g(\sqcup S)$.

Alternatively, a more intuitive solution: If g were continuous, it would be computable. As it implicitly solves the halting program for an input function, it is known to be uncomputable, hence, we have a contradiction.

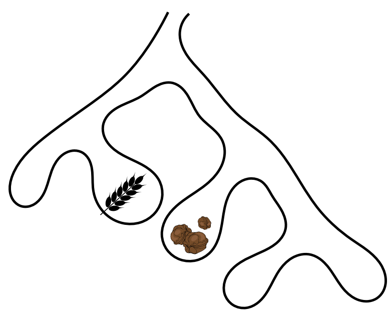
d) **No**. Consider the term $(\lambda x.42)$ `bot` as example (with the usual δ-rule `bot` $\to$ `bot` for `bot`), which is reduced to 42 using an outermost strategy and does not have a normal form when reducing according to an innermost strategy.

e) **Yes**, as Simple Haskell can be implemented using the $\to_{\beta\delta}$ reduction.

Exercise 2 (Programming in Haskell): (10 + 10 + 8 + 10 + 6 = 44 points)

We define a polymorphic data structure `HamsterCave` to represent hamster caves which can contain different types of food.

```
data HamsterCave food
    = EmptyTunnel
    | FoodTunnel food
    | Branch (HamsterCave food) (HamsterCave food)
    deriving Show
```



The data structure `HamsterFood` is used to represent food for hamsters. For example, `exampleCave` is a valid expression of type `HamsterCave HamsterFood`.

```
data HamsterFood = Grain | Nuts deriving Show

exampleCave :: HamsterCave HamsterFood
exampleCave = Branch
    (Branch EmptyTunnel (FoodTunnel Grain))
    (Branch (FoodTunnel Nuts) (Branch EmptyTunnel EmptyTunnel))
```

a) Implement a function `digAndFillCave :: Int -> HamsterCave HamsterFood`, such that for any integer number `n` > 1, `digAndFillCave n` creates a hamster cave without empty tunnels of depth `n`, such that the number of `FoodTunnels` containing `Grain` equals the number of `FoodTunnels` containing `Nuts`. Here, the depth of a cave is the maximal number of "nodes" on any path from the entry of the cave to a dead end. Thus, `exampleCave` has depth 4.

b) Implement a fold function `foldHamsterCave`, including its type declaration, for the data structure `HamsterCave`. As usual, the fold function replaces the data constructors in a `HamsterCave` expression by functions specified by the user. The first argument of `foldHamsterCave` should be the function for the case of the empty tunnel, the second argument the function for the case of the food tunnel, and the third argument the function for the case of a branch. As an example, the following function definition uses `foldHamsterCave` to determine the number of dead ends (either with or without food) in a cave, such that the call `numberOfDeadEnds exampleCave` returns 5.

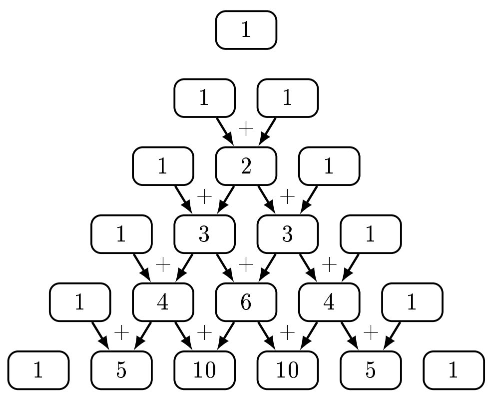
```
numberOfDeadEnds :: HamsterCave food -> Int
numberOfDeadEnds cave = foldHamsterCave 1 (\_ -> 1) (+) cave
```

c) Implement the function `collectFood :: HamsterCave food -> (HamsterCave food, [food])`, which returns a tuple for a given hamster cave. The first argument of the tuple is the same hamster cave as the one given to the function, but without any food (i.e., every `FoodTunnel` is replaced by an `EmptyTunnel`). The second argument is a list of all the food that was removed from the cave. For the definition of `collectFood`, use only one defining equation where the right-hand side is a call to the function `foldHamsterCave`.

For example, a call `collectFood exampleCave` should return the following tuple:

```
(Branch (Branch EmptyTunnel EmptyTunnel)
        (Branch EmptyTunnel (Branch EmptyTunnel EmptyTunnel))
,[Grain,Nuts])
```

d) Implement a cyclic data structure `pascalsTriangle :: [[Int]]` (consisting of lists of lists of `Ints`) that represents Pascal's triangle. The first row of the triangle is represented by the first list of integers (`[1]`), the second row by the second list (`[1,1]`), and so forth. Each row in Pascal's triangle is constructed from its preceding row, by adding each pair of consecutive numbers. For this, it is assumed that all numbers lying outside of the preceding row are zeros.
Hint: You should use use the function `zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]`, which applies the function given as its first argument to combine the elements of two lists. For example `zipWith (++) ["a","b"] ["c", "d", "e"]` results in the list `["ac","bd"]`. Note that the length of the resulting list is the smallest length of both input lists.



e) Write a Haskell expression in form of a *list comprehension* to compute all prime numbers. To determine if a number `i` is prime, test whether no number from `2` to `i - 1` divides `i`. You may use the functions `all :: (a -> Bool) -> [a] -> Bool` where `all p xs` is `True` iff `p x` is `True` for all elements `x` of the list `xs`, the function `not :: Bool -> Bool`, and the function `divides` as defined below.

```
divides :: Int -> Int -> Bool
i `divides` j = j `mod` i == 0
```

Solution:

a)
```
digAndFillCave :: Int -> HamsterCave HamsterFood
digAndFillCave n | n > 1 = Branch (cave Nuts (n-1)) (cave Grain (n-1))
  where
    cave food 1 = FoodTunnel food
    cave food n = Branch (cave food (n-1)) (cave food (n-1))
```

b)
```
foldHamsterCave
    :: result
    -> (food -> result)
    -> (result -> result -> result)
    -> HamsterCave food
    -> result
foldHamsterCave fET fTWF fTB = go
  where
    go EmptyTunnel = fET
    go (FoodTunnel f) = fTWF f
    go (Branch left right) = fTB (go left) (go right)
```

c)
```
collectFood :: HamsterCave food -> (HamsterCave food, [food])
collectFood = foldHamsterCave
        (EmptyTunnel, [])
        (\x -> (EmptyTunnel, [x]))
        (\(tl, fl) (tr, fr) -> (Branch tl tr, fl ++ fr))
```

d)
```
pascalsTriangle :: [[Int]]
pascalsTriangle = [1] : map nextRow pascalsTriangle
  where
    nextRow oldRow = zipWith (+) (oldRow ++ [0]) ([0] ++ oldRow)
```

e)
```
[ i | i <- [2..], all (\j -> not (j `divides` i)) [2..i-1] ]
```

Exercise 3 (Semantics): (21 + 10 + 5 + 4 = 40 points)

a) i) Let $\sqsubseteq$ be a cpo on D and $f : D \to D$ be continuous. Prove the fixpoint theorem, i.e., that $\sqcup\{f^i(\perp) \mid i \in \mathbb{N}\}$ exists and that this is the least fixpoint of f. You may use all other results from the lecture in your proof.

ii) Let $D = 2^{\mathbb{N}}$, i.e., D is the set of all sets of natural numbers and let $\subseteq$ denote the usual subset relation.

1) Prove that every chain $S \subseteq D$ has a least upper bound w.r.t. the relation $\subseteq$.

2) Prove that $\subseteq$ is a cpo on D.

3) Give an example for an infinite chain in $(D, \subseteq)$.

4) Give a monotonic, non-continuous function $f : D \to D$. You do not need to prove that f has these properties.

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

```
mult :: (Int, Int) -> Int
mult (0, y) = 0
mult (x, y) = y + mult (x - 1, y)
```

Please give the Haskell declaration for the higher-order function `f_mult` corresponding to `mult`, i.e., the higher-order function `f_mult` such that the least fixpoint of `f_mult` is `mult`. In addition to the function declaration, please also give the type declaration of `f_mult`. You may use full Haskell for `f_mult`.

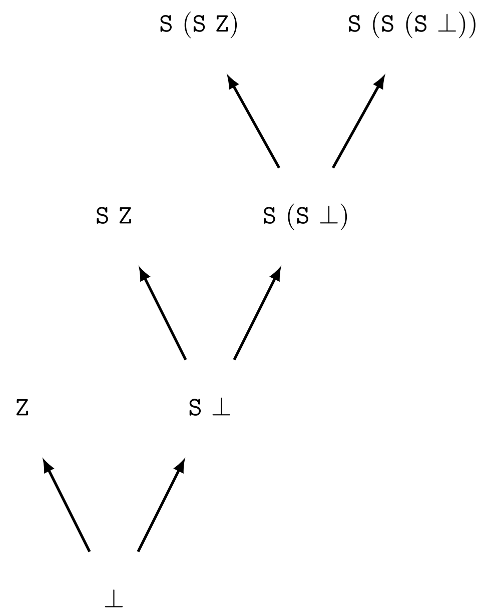
ii) Let $\phi_{\texttt{f\_mult}}$ be the semantics of the function `f_mult`. Give the semantics of $\phi^n_{\texttt{f\_mult}}(\bot)$ for $n \in \mathbb{N}$, i.e., the semantics of the n-fold application of $\phi_{\texttt{f\_mult}}$ to $\bot$.

iii) Give all fixpoints of $\phi_{\texttt{f\_mult}}$ and mark the least fixpoint.

c) Consider the following data type declaration for natural numbers:

```
data Nats = Z | S Nats
```

A graphical representation of the first four levels of the domain for `Nats` could look like this:



Now consider the following data type declarations:

```
data X = A X Y | B Y
data Y = E Y | H
```

Give a graphical representation of the first three levels of the domain for the type `X`. The third level contains the element `A (A ⊥ ⊥) ⊥`, for example.

d) Consider the usual definition for `Nats` above, i.e., `data Nats = Z | S Nats`.

Write a function `plus :: Nats -> Nats -> Nats` in **Simple** Haskell that computes the sum of two natural numbers, i.e., `plus S(S(Z)) S(Z)` should yield `S(S(S(Z)))`. Your solution should use the functions defined in the transformation from the lecture such as $\texttt{sel}_{n,i}$, $\texttt{isa}_{\underline{\texttt{constr}}}$, $\texttt{argof}_{\underline{\texttt{constr}}}$, and `bot`. You do not have to use the transformation rules from the lecture, though.

Solution:

a) i) We first prove that $f^i(\bot) \sqsubseteq f^{i+1}(\bot)$ holds for all $i \in \mathbb{N}$ by induction. As base case, we consider $i = 0$ and of course, $f^0(\bot) = \bot \sqsubseteq f^1(\bot)$ holds.

In the induction step, we assume that for some $i > 0$, $f^{i-1}(\bot) \sqsubseteq f^i(\bot)$ holds. Then, because f is continuous, f is also monotonic, hence $f(f^{i-1}(\bot)) \sqsubseteq f(f^i(\bot)) \Leftrightarrow f^i(\bot) \sqsubseteq f^{i+1}(\bot)$ holds.

Thus, $\{f^i(\bot) \mid i \in \mathbb{N}\}$ is a chain and because $\sqsubseteq$ is a cpo on D, $\sqcup\{f^i(\bot) \mid i \in \mathbb{N}\}$ exists. We now need to prove that this is the least fixpoint of f. First, we prove that this is indeed a fixpoint:

$$\begin{aligned}
f(\sqcup\{f^i(\bot) \mid i \in \mathbb{N}\}) &= \sqcup f(\{f^i(\bot) \mid i \in \mathbb{N}\}) && (f \text{ continuous})\\
&= \sqcup\{f^{i+1}(\bot) \mid i \in \mathbb{N}\}\\
&= \sqcup(\{f^{i+1}(\bot) \mid i \in \mathbb{N}\} \cup \{\bot\})\\
&= \sqcup\{f^i(\bot) \mid i \in \mathbb{N}\}
\end{aligned}$$

Now assume there is another fixpoint d of f. We need to prove $\sqcup\{f^i(\bot) \mid i \in \mathbb{N}\} \sqsubseteq d$ and do this by inductively proving $f^i(\bot) \sqsubseteq d$. In the base case, $f^0(\bot) = \bot \sqsubseteq d$ obviously holds. In the induction step, assume $f^i(\bot) \sqsubseteq d$ already holds. Then, because f is monotonic, we have $f(f^i(\bot)) \sqsubseteq f(d)$. But as d is a fixpoint of f, we can conclude that $f^{i+1}(\bot) \sqsubseteq d$.

ii) Let $S = \{M_1, M_2, \ldots\}$ with $M_i \subseteq M_{i+1}$.

1) We have $\sqcup S = \bigcup M_i$. Obviously, $M_i \subseteq \bigcup M_i$. Now assume that there is some other upper bound B with $\bigcup M_i \not\subseteq B$. Then there is some $e \in \bigcup M_i \setminus B$ and by construction, there is some k with $e \in M_k$. As $e \notin B$, we have $M_k \not\subseteq B$ and hence, B is not an upper bound of S w.r.t. $\subseteq$. Thus, we have a contradiction.

2) In 1), we have proven that for every chain, there exists a lub. Obviously, we have $\bigcup M_i \in D$. With $\emptyset$ as the minimal element, $\subseteq$ is a cpo for D.

3) Let $N_i := \{k \in \mathbb{N} \mid k \leq i\}$. Then, $N_i \subseteq N_{i+1}$ holds and hence, $\{N_1, N_2, \ldots\}$ is a chain.

4)

$$f(M) = \begin{cases} \emptyset & M \text{ is finite} \\ \{42\} & \text{otherwise} \end{cases}$$

Alternative: The function g from Ex. 1 c).

b) i)

```
f_mult :: ((Int, Int) -> Int) -> ((Int, Int) -> Int)
f_mult mult (0, y) = 0
f_mult mult (x, y) = y + mult (x - 1, y)
```

ii)

$$(\phi^n_{\texttt{f\_mult}}(\bot))(x, y) = \begin{cases} 0 & \text{if } x = 0 \wedge n > 0 \\ x \cdot y & \text{if } 0 < x < n \wedge y \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

iii) The least fixpoint of $\phi_{\texttt{f\_mult}}$ is the function

$$g(x, y) = \begin{cases} 0 & \text{if } x = 0 \\ x \cdot y & \text{if } 0 < x \wedge y \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

Another fixpoint is the function

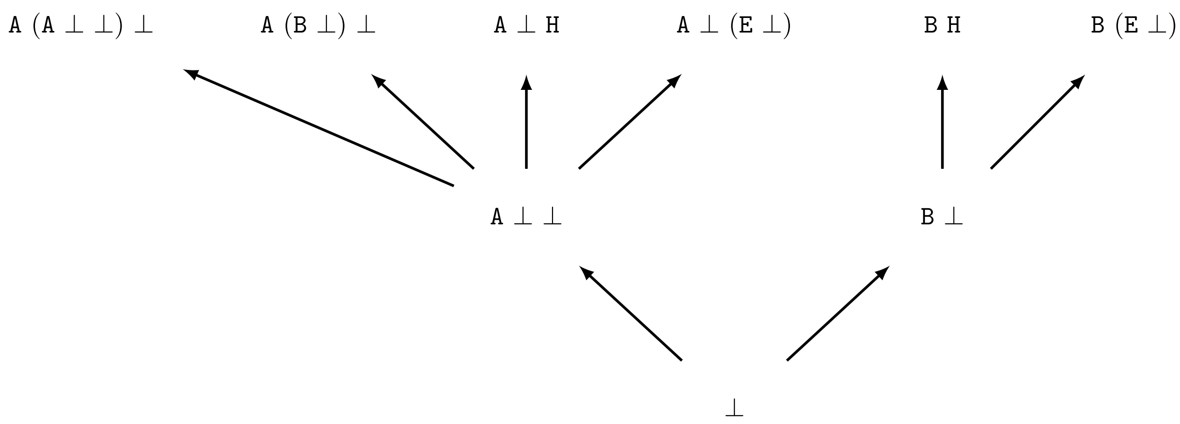
$$h(x, y) = \begin{cases} 0 & \text{if } x = 0 \\ x \cdot y & \text{if } x \neq \bot \wedge y \neq \bot \\ \bot & \text{if } x = \bot \vee (x \neq 0 \wedge y = \bot) \quad \text{(this is “otherwise”)} \end{cases}$$

To be a fixpoint, a function f has to satisfy the equality $f(c_1, c_2) = \phi_{\texttt{f\_mult}}(f)(c_1, c_2)$, which is equivalent to $f(c_1, c_2) = 0$ for $c_1 = 0$ (this is the first case in the definitions above).

For $c_1 \neq 0$, we have $f(c_1, c_2) = c_2 + f(c_1 - 1, c_2)$. This implies that for $c_1 = \bot$, the result has to be $\bot$, as $c_1 - 1$ is not well-defined in that case. For $c_2 = \bot$ (and $c_1 \neq 0$, as that case was handled above), the result also has to be $\bot$, as $c_2 + f(c_1 - 1, c_2)$ is not well-defined in that case. This corresponds to the last case in the definition of h.

So finally, we are left with the cases for $c_1, c_2 \in \mathbb{Z}$, for which $f(c_1, c_2) = c_2 + f(c_1 - 1, c_2)$ has to hold, which is exactly the condition for multiplication, yielding the middle case.

c)



d)
```
plus = \x -> \y ->
          if (isa_Z x) then y
          else if (isa_S x)
               then S (plus (argof_S x) y)
               else bot
```

Alternative:

```
plus = \x -> \y ->
          if (isa_Z x) then y
          else S (plus (argof_S x) y)
```

Exercise 4 (Lambda Calculus): (4 + 6 = 10 points)

a) Please translate the following Haskell expression into an equivalent lambda term (e.g., using $\mathcal{L}am$). Translate the pre-defined function `<` to `LessThan`, `+` to `Plus` and `-` to `Minus` (remember that the infix notation of `<`, `+`, `-` is not allowed in lambda calculus). It suffices to give the result of the transformation:

```
let quot = \x y -> if x < y then 0 else 1 + quot (x-y) y in quot v w
```

b) Let $t = \lambda \mathit{fact}.(\lambda x.(\texttt{If}\ (\texttt{LessThanOrE}\ x\ 1)\ 1\ (\texttt{Times}\ x\ (\mathit{fact}\ (\texttt{Minus}\ x\ 1)))))$ and

$$
\begin{aligned}
\delta = \; & \{\ \texttt{If True} \to \lambda x\ y.x, \\
& \quad \texttt{If False} \to \lambda x\ y.y, \\
& \quad \texttt{fix} \to \lambda f.f(\texttt{fix}\ f)\} \\
\cup \; & \{\ \texttt{Minus}\ x\ y \to z \mid x, y \in \mathbb{Z} \wedge z = x - y\} \\
\cup \; & \{\ \texttt{Times}\ x\ y \to z \mid x, y \in \mathbb{Z} \wedge z = x \cdot y\} \\
\cup \; & \{\ \texttt{LessThanOrE}\ x\ y \to b \mid x, y \in \mathbb{Z} \wedge ((x \leq y \wedge b = \texttt{True}) \vee (x > y \wedge b = \texttt{False}))\}
\end{aligned}
$$

Please reduce $\texttt{fix}\ t\ 1$ 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 the term it represents whenever possible.

Solution:

a) $(\texttt{fix}\ (\lambda quot\ x\ y.\texttt{If}\ (\texttt{LessThan}\ x\ y)\ 0\ (\texttt{Plus}\ 1\ (quot\ (\texttt{Minus}\ x\ y)\ y))))\ v\ w$

b)

$$\begin{array}{ll}
 & \texttt{fix}\ t\ 1 \\
\rightarrow_\delta & (\lambda f.(f\ (\texttt{fix}\ f)))\ t\ 1 \\
\rightarrow_\beta & t\ (\texttt{fix}\ t)\ 1 \\
\rightarrow_\beta & (\lambda x.(\texttt{If}\ (\texttt{LessThanOrE}\ x\ 1)\ 1\ (\texttt{Times}\ x\ (\texttt{fix}\ t\ (\texttt{Minus}\ x\ 1)))))\ 1 \\
\rightarrow_\beta & \texttt{If}\ (\texttt{LessThanOrE}\ 1\ 1)\ 1\ (\texttt{Times}\ 1\ (\texttt{fix}\ t\ (\texttt{Minus}\ 1\ 1))) \qquad (*) \\
\rightarrow_\delta & \texttt{If}\ \texttt{True}\ 1\ (\texttt{Times}\ 1\ (\texttt{fix}\ t\ (\texttt{Minus}\ 1\ 1))) \\
\rightarrow_\delta & (\lambda x.(\lambda y.x))\ 1\ (\texttt{Times}\ 1\ (\texttt{fix}\ t\ (\texttt{Minus}\ 1\ 1))) \\
\rightarrow_\beta & (\lambda y.1)\ (\texttt{Times}\ 1\ (\texttt{fix}\ t\ (\texttt{Minus}\ 1\ 1))) \\
\rightarrow_\beta & 1
\end{array}$$

[The original exam had a mixed use of `If` and `if`, so technically, it was OK to stop after reaching the term marked with $(*)$.]

Exercise 5 (Type Inference): (6 points)

Using the initial type assumption $A_0 := \{x :: \forall a.a \rightarrow \texttt{Int}\}$ infer the type of the expression $\lambda y.y\,x$ using the algorithm $\mathcal{W}$.

Solution:

$$\begin{array}{l}
\mathcal{W}(A_0, \lambda y.y\,x) \\
\quad \mathcal{W}(A_0 + \{y :: b_1\}, y\,x) \\
\quad\quad \mathcal{W}(A_0 + \{y :: b_1\}, y) = (id, b_1) \\
\quad\quad \mathcal{W}(A_0 + \{y :: b_1\}, x) = (id, b_2 \rightarrow \texttt{Int}) \\
\quad mgu(b_1, (b_2 \rightarrow \texttt{Int}) \rightarrow b_3) = [b_1/(b_2 \rightarrow \texttt{Int}) \rightarrow b_3]) \\
\quad = ([b_1/(b_2 \rightarrow \texttt{Int}) \rightarrow b_3], b_3) \\
= ([b_1/(b_2 \rightarrow \texttt{Int}) \rightarrow b_3], ((b_2 \rightarrow \texttt{Int}) \rightarrow b_3) \rightarrow b_3)
\end{array}$$