Solutions

1. Consider the λ-term $S = \lambda x\, y\, z. x\ z\ (y\ z)$

[12] (a) Reduce the term $S\ S\ S$ to normal form, using the leftmost innermost reduction strategy.

Solution.

$$
\begin{aligned}
& S\ S\ S \\
&= (\lambda x\ y\ z.x\ z\ (y\ z))\ (\lambda x\ y\ z.x\ z\ (y\ z))\ (\lambda x\ y\ z.x\ z\ (y\ z)) \\
&\to_\beta (\lambda y\ z.(\lambda x\ y\ z.x\ z\ (y\ z))\ z\ (y\ z))\ (\lambda x\ y\ z.x\ z\ (y\ z)) \\
&\to_\beta (\lambda y\ z.(\lambda y\ z_1.z\ z_1\ (y\ z_1))\ (y\ z))\ (\lambda x\ y\ z.x\ z\ (y\ z)) \\
&\to_\beta (\lambda y\ z\ z_1.z\ z_1\ (y\ z\ z_1))\ (\lambda x\ y\ z.x\ z\ (y\ z)) \\
&\to_\beta \lambda z\ z_1.z\ z_1\ ((\lambda x\ y\ z.x\ z\ (y\ z))\ z\ z_1) \\
&\to_\beta \lambda z\ z_1.z\ z_1\ ((\lambda y\ z_1.z\ z_1\ (y\ z_1))\ z_1) \\
&\to_\beta \lambda z\ z_1.z\ z_1\ (\lambda z_2.z\ z_2\ (z_1\ z_2))
\end{aligned}
$$

[13] (b) Reduce the term $S\ S\ S$ to normal form, using the leftmost outermost reduction strategy.

Solution.

$$
\begin{aligned}
& S\ S\ S \\
&= (\lambda x\ y\ z.x\ z\ (y\ z))\ (\lambda x\ y\ z.x\ z\ (y\ z))\ (\lambda x\ y\ z.x\ z\ (y\ z)) \\
&\to_\beta (\lambda y\ z.(\lambda x\ y\ z.x\ z\ (y\ z))\ z\ (y\ z))\ (\lambda x\ y\ z.x\ z\ (y\ z)) \\
&\to_\beta \lambda z.(\lambda x\ y\ z.x\ z\ (y\ z))\ z\ ((\lambda x\ y\ z.x\ z\ (y\ z))\ z) \\
&\to_\beta \lambda z.(\lambda y\ z_1.z\ z_1\ (y\ z_1))\ ((\lambda x\ y\ z.x\ z\ (y\ z))\ z) \\
&\to_\beta \lambda z\ z_1.z\ z_1\ ((\lambda x\ y\ z.x\ z\ (y\ z))\ z\ z_1) \\
&\to_\beta \lambda z\ z_1.z\ z_1\ ((\lambda y\ z_1.z\ z_1\ (y\ z_1))\ z_1) \\
&\to_\beta \lambda z\ z_1.z\ z_1\ (\lambda z_2.z\ z_2\ (z_1\ z_2))
\end{aligned}
$$

2. Consider the three OCaml functions

```
let rec take n xs = if n < 1 then [] else match xs with
  | [] -> []
  | x::xs -> x :: take (n-1) xs

let rec length = function [] -> 0
                        | _::xs -> 1 + length xs

let rec init = function x::y::xs -> x :: init (y::xs)
                      | _ -> []
```

Prove by induction that `init` $xs =$ `take` (`length` $xs - 1$) xs for every list xs. (*Hint:* In the step case, you will need a further case distinction on the tail of the list.)

[5] (a) Base case.

Solution.

Base Case ($xs =$ `[]`). By applying the definitions of the three functions, we prove the base case as follows: `take` (`length` `[]` $- 1$) `[]` $=$ `take` (-1) `[]` $=$ `[]` $=$ `init` `[]`.

[20] (b) Step case.

Solution.

Step Case ($xs = z :: zs$). The IH is that init zs = take (length $zs - 1$) zs. Since the first pattern in the definition of init requires at least two elements in xs, we do a further case distinction on zs.

Case 1 (zs = [])

$$\begin{aligned}\texttt{init}\ (z :: \texttt{[]}) &= \texttt{[]}\\ &= \texttt{take}\ 0\ (z :: \texttt{[]})\\ &= \texttt{take}\ (\texttt{length}\ (z :: \texttt{[]}) - 1)\ (z :: \texttt{[]})\end{aligned}$$

Case 2 ($zs = w :: ws$)

$$\begin{aligned}\texttt{init}\ (z :: w :: ws) &= z :: \texttt{init}\ (w :: ws)\\ &= z :: \texttt{take}\ (\texttt{length}\ zs - 1)\ zs && \text{by IH}\\ &= \texttt{take}\ (\texttt{length}\ (z :: zs) - 1)\ (z :: zs)\end{aligned}$$

3. Consider the OCaml function:

```
let rec sum = function [] -> 0
                     | x::xs -> x + sum xs
```

[12] (a) Implement a tail-recursive variant of sum.

Solution.

```
let sum xs =
  let rec sum' acc = function [] -> acc
                            | x::xs -> sum' (x + acc) xs
  in
  sum' 0 xs
```

[13] (b) Use tupling to implement a function average : int list -> int, producing the same results as if defined via average xs = sum xs/length xs.

Solution.

```
let average xs =
  let rec average' s l = function [] -> (s, l)
                                | x::xs -> average' (x + s) (l + 1) xs
  in
  let (s, l) = average' 0 0 xs in
  s / l
```

4. Consider the typing environment $E = \{\mathsf{true} : \mathsf{bool}\}$.

[12] (a) Use type checking to decide whether the expression **let** x = true **in** $x\ x$ is of type bool with respect to the environment E. Justify your answer.

Solution. By the rule (let), we need to be able to construct a proof tree for $E, x : \mathsf{bool} \vdash x\ x : \mathsf{bool}$. Since x is not of function type, this is impossible.

[13] (b) Solve (if possible) the unification problem:

$$\alpha_1 \to \alpha_2 \to \alpha_3 \approx \alpha_4 \to (\alpha_2 \to \alpha_2) \to \alpha_5$$

Solution. After two applications of rule (d_2), we obtain:

$$\begin{aligned}\alpha_1 &\approx \alpha_4\\ \alpha_2 &\approx \alpha_2 \to \alpha_2\\ \alpha_3 &\approx \alpha_5\end{aligned}$$

Now, no rule is applicable to the second equation and thus there is no solution.