1 Hacking with the untyped call-by-value lambda calculus

In this exercise, you have to implement some operations for Church encoding of lists. There are several ways to Church encode a list, among which Church encoding based on its right fold function is more popular. As an example, an empty list (`nil`) and the `cons` construct are represented as follows in this encoding:

```
nil = λc.  λn.  n
cons = λh.  λt.  λc.  λn.  c h (t c n)
```

As another example, a list of 3 elements x, y, z is encoded as:

```
λc.  λn.  c x (c y (c z n))
```

The complete list of predefined operations can be found in the appendix, and only these operations can be used in the exercise. Define the following operations on a list:
(For explanations of the solutions, see the subsection below the questions)

1. (2 points) The *map* function which applies the given function to each element of the given list.

   ```
   map = λ f. λ l. l (λ h. λ r. cons (f h) r) nil
   ```

2. (2 points) The *length* function which returns the size of the given list. The result should be in Church encoding.

   `length = λl. l (λa. λb. scc b)` c_0

3. (2 points) The *sum* function which returns the sum of all elements of the given list. Assume all elements and the result are Church encoded numbers.

   `sum = λl. l plus` c_0

4. (2 points) The *concat* function which concatenates two input lists.

   ```
   concat = λl1. λl2. l1 cons l2
   ```

5. (2 points) The *exists* function which checks if there is any element satisfying the given predicate. The given predicate and the result should be both in Church encoding.

   ```
   exists = λl. λp. l (λa. λb. p a tru b) fls
   ```

1.1 Explanations

This task is much easier if you understand how the encoding works. We say it is the "right fold" not without accident. Basically, lists in this encoding work like partial application of the `foldRight` method – `cons 1 (cons 2 nil)` is like `List(1,2).foldRight`.

In Scala, function calls like `List(1,2).foldRight(0)(_ + _)` are equivalent to `1 + (2 + 0)` – notice how `foldRight` inserts the folding function between every element of the list, puts 0 at the end, and associates operations to the right. The same thing goes for this encoding. If we have `l = cons 1 (cons 2 nil)`, then `l f z = f 1 (f 2 z)` – observe how we basically replaced `cons` with `f` and `nil` with `z`.

Get a feeling for how this encoding works! You might see something similar during the exam.

2 Simply typed SKI combinators

In this exercise we're going to explore an alternative calculus called SKI that's based on three combinators: S, K and I instead of lambda abstraction. Those combinators can be translated into STLC as derived forms:

$$I[T] = \lambda x : T.\ x \tag{D-I}$$

$$K[T, U] = \lambda x : T.\ \lambda y : U.\ x \tag{D-K}$$

$$S[T, U, W] = \lambda x : T \to U \to W.\ \lambda y : T \to U.\ \lambda z : T.\ xz(yz) \tag{D-S}$$

An interesting aspect of SKI is that those combinators are sufficient to exclude lambda abstraction from the language without loss of expressiveness. More concretely the system has the following syntax:

$$\begin{array}{lcll}
t & ::= & & \textbf{terms}: \\
 & | & I[T] & I \text{ combinator} \\
 & | & K[T,U] & K \text{ combinator} \\
 & | & S[T,U,W] & S \text{ combinator} \\
 & | & t\ t & \text{Application} \\
u, v, w & ::= & & \textbf{values}: \\
 & | & I[T] & I \text{ combinator} \\
 & | & K[T,U] & K \text{ combinator (1)} \\
 & | & K[T,U]\ v & K \text{ combinator (2)} \\
 & | & S[T,U,W] & S \text{ combinator (1)} \\
 & | & S[T,U,W]\ v & S \text{ combinator (2)} \\
 & | & S[T,U,W]\ v\ v & S \text{ combinator (3)} \\
T, U, V, W & ::= & & \textbf{types}: \\
 & | & T \to T & \text{Function type}
\end{array}$$

Values in this language are the aforementioned combinators as well as their partially applied versions.

Questions:

- Provide small-step reduction rules assuming call-by-value evaluation semantics (4 points).
- Provide typing rules $\Gamma \vdash t : T$ and prove the *preservation* property (6 points).

Note: There is no lambda abstraction in the language any longer. You may not use it as a means to express typing or evaluation rules.

2.1 Solution

Small-step reduction rules:

$$I[T]\ v \longrightarrow v \tag{R-I}$$

$$K[T, U]\ v_1\ v_2\ \longrightarrow v_1 \tag{R-K}$$

$$S[T,U,W]\ v_1\ v_2\ v_3\ \longrightarrow v_1 v_3 (v_2 v_3) \tag{R-S}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \tag{R-App1}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \tag{R-App2}$$

Typing rules:

$$\Gamma \vdash I[T] : T \to T \tag{T-I}$$

$$\Gamma \vdash K[T,U] : T \to U \to T \tag{T-K}$$

$$\Gamma \vdash S[T,U,W] : (T \to U \to W) \to (T \to U) \to T \to W \tag{T-S}$$

$$\frac{\Gamma \vdash t_1 : T_2 \to T \quad \Gamma \vdash t_2 : T_2}{\Gamma \vdash t_1\ t_2 : T} \tag{T-App}$$

Theorem 1 (Preservation). *If $\Gamma \vdash t : T$ and $t \longrightarrow t'$, then $\Gamma \vdash t' : T$.*

Proof. The proof is by induction on reduction derivations and case analysis on typing derivations.

Case (R-I). We have $t = I[V]\ v$, $t' = v$ and $\Gamma \vdash I[V]\ v : T$.

The only typing rule that applies is (T-App), so we have $\Gamma \vdash I[V] : V' \to T$ and $\Gamma \vdash v : V'$ for some V'. By case-analysis on typing derivations, we find that the only rule that could have been used to derive $\Gamma \vdash I[V] : V' \to T$ is (T-I), and hence $V' = T$. Therefore $\Gamma \vdash v : T$, and we are done.

Case (R-K). We have $t = K[V,W]\ v\ w$, $t' = v$ and $\Gamma \vdash K[V,W]\ v\ w : T$.

This case is similar to that for (R-K): again, the only typing rule that applies is (T-App) and by repeated case-analysis, we find

- from (T-App), that $\Gamma \vdash K[V,W]\ v : W' \to T$ and $\Gamma \vdash w : W'$ for some W' ,
- from (T-App), that $\Gamma \vdash K[V,W] : V' \to W' \to T$ and $\Gamma \vdash v : V'$ for some V',
- from (T-K), that $\Gamma \vdash K[V,W] : V \to W \to V$, so $V' = V = T$ and $W' = W$.

Hence $\Gamma \vdash v : T$, and we are done.

Case (R-S). We have $t = S[U,V,W]\ u\ v\ w$, $t' = u\ w\ (v\ w)$ and $\Gamma \vdash S[U,V,W]\ u\ v\ w : T$.

As for the cases of (R-I) and (R-K), we find, by repeated case-analysis on the typing derivation of $\Gamma \vdash S[U,V,W]\ u\ v\ w : T$, that

- $\Gamma \vdash S[U,V,W] : (U \to V \to W) \to (U \to V) \to U \to W$ with $W = T$,
- $\Gamma \vdash u : U \to V \to W$,
- $\Gamma \vdash v : U \to V$, and
- $\Gamma \vdash w : U$.

By repeated application of (T-App), it follows that $\Gamma \vdash u\ w\ (v\ w) : T$.

Case (R-App1). We have $t = t_1\ t_2$, $t' = t_1'\ t_2$, $t_1 \longrightarrow t_1'$ and $\Gamma \vdash t_1\ t_2 : T$.

Again, the only typing rule that applies is (T-App), so we have $\Gamma \vdash t_1 : U \to T$ and $\Gamma \vdash t_2 : U$ for some U. By the IH, $\Gamma \vdash t_1' : U \to T$, and by (T-App), $\Gamma \vdash t_1'\ t_2 : T$.

Case (R-App2). Similar to the previous case.

$\square$

3 Appendix

3.1 The call-by-value simply typed lambda calculus

The complete reference of the variant of simply typed lambda calculus (with *Bool* ground type representing the type of values *true* and $false$) used in "The call-by-value simply typed lambda calculus with returns" is as follows:

$$\begin{array}{ll}
v ::= \lambda x\colon T.\ t \mid bv & (\textit{values}) \\
bv ::= \texttt{true} \mid \texttt{false} & (\textit{boolean values}) \\
t ::= x \mid v \mid t\ t & (\textit{terms}) \\
p ::= t & (\textit{programs}) \\
T ::= \texttt{Bool} \mid T \to T & (\textit{types})
\end{array}$$

Evaluation rules:

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \qquad \text{(E-App1)}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \qquad \text{(E-App2)}$$

$$(\lambda x\colon T_1.\ t_1)\ v_2 \longrightarrow [x \mapsto v_2]t_1 \qquad \text{(E-AppAbs)}$$

Typing rules:

$$\frac{x\colon T \in \Gamma}{\Gamma \vdash x\colon T} \qquad \text{(T-Var)}$$

$$\frac{\Gamma,\ x\colon T_1 \vdash t_2 \ :\ T_2}{\Gamma \vdash (\lambda x\colon T_1.\ t_2) \ :\ T_1 \to T_2} \qquad \text{(T-Abs)}$$

$$\frac{\Gamma \vdash t_1 \ :\ T_1 \to T_2 \quad \Gamma \vdash t_2 \ :\ T_1}{\Gamma \vdash t_1\ t_2 \ :\ T_2} \qquad \text{(T-App)}$$

$$\frac{}{\Gamma \vdash \texttt{true} \ :\ \texttt{Bool}} \qquad \text{(T-False)}$$

$$\frac{}{\Gamma \vdash \texttt{false} \ :\ \texttt{Bool}} \qquad \text{(T-True)}$$