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. (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) c0
   ```

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 c0
   ```

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:

t	$::=$		**terms** :
	$\mid$	$I[T]$	I combinator
	$\mid$	$K[T, U]$	K combinator
	$\mid$	$S[T, U, W]$	S combinator
	$\mid$	$t\ t$	Application
u, v, w	$::=$		**values** :
	$\mid$	$I[T]$	I combinator
	$\mid$	$K[T, U]$	K combinator (1)
	$\mid$	$K[T, U]\ v$	K combinator (2)
	$\mid$	$S[T, U, W]$	S combinator (1)
	$\mid$	$S[T, U, W]\ v$	S combinator (2)
	$\mid$	$S[T, U, W]\ v\ v$	S combinator (3)
T, U, V, W	$::=$		**types** :
	$\mid$	$T \to T$	Function type

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.

□

3 Checked Error Handling

In this exercise we use the Simply-Typed Lambda Calculus (STLC) extended with rules for error handling. In this language, terms may reduce to a normal form `error`, which is *not* a value. In addition, we add the new term form `try` t_1 `with` t_2, which allows handling errors that occur while evaluating t_1.

Here is a summary of the extensions to syntax and evaluation:

$$
\begin{array}{rcll}
t & ::= & & \textbf{terms}: \\
& | & \ldots & \\
& | & \texttt{error} & \text{run-time error} \\
& | & \texttt{try } t \texttt{ with } t & \text{trap errors}
\end{array}
$$

New evaluation rules:

$$\text{(E-AppErr1)}\ \texttt{error}\ t_2 \longrightarrow \texttt{error} \qquad \text{(E-AppErr2)}\ v_1\ \texttt{error} \longrightarrow \texttt{error}$$

$$\text{(E-TryValue)}\ \texttt{try}\ v_1\ \texttt{with}\ t_2 \longrightarrow v_1 \qquad \text{(E-TryError)}\ \texttt{try error with}\ t_2 \longrightarrow t_2$$

$$\text{(E-Try)}\ \frac{t_1 \longrightarrow t_1'}{\texttt{try}\ t_1\ \texttt{with}\ t_2 \longrightarrow \texttt{try}\ t_1'\ \texttt{with}\ t_2}$$

(Note that these extensions are exactly those summarized in Figures 14-1 and 14-2 on pages 172 and 174 of the TAPL book. However, also note that we will use *different* typing rules.)

The goal of this exercise is to define typing rules for STLC with the above extensions such that the following progress theorem holds:

$$\text{If } \emptyset\ ;\ \texttt{false} \vdash t : T, \text{ then either } t \text{ is a value or else } t \longrightarrow t'.$$

The above theorem uses a typing judgment extended with a Boolean value E, written $\Gamma\ ;\ E \vdash t : T$ where $E \in \{\texttt{true}, \texttt{false}\}$. The theorem says that a well-typed term that is closed (that is, it does not have free variables, which is expressed using $\Gamma = \emptyset$) is either a value, or else it can be reduced *as long as* $E = \texttt{false}$.

Your task is to find out how the value of E can be used to distinguish the terms that may reduce to `error` from those terms that may never reduce to `error`. Note that `error` is a normal form, but it is *not* a value.

1. Specify typing rules of the form $\Gamma\ ;\ E \vdash t : T$ for all term forms of STLC with the above extensions such that the above progress theorem holds.

2. Prove the above progress theorem using structural induction. (You can use the canonical forms lemma for STLC as seen in the lecture without proof.)

3.1 Solution

Attention: Preservation doesn't hold for this solution, unless we introduce effect annotation on function types.

Typing rules. Intuitively, the predicate E in the typing judgment $\Gamma\ ;\ E \vdash t : T$ determines whether the term t is in an *impure* position, i.e. whether it is allowed to reduce to `error` or not. The typing rules for the error (handling) terms are as follows:

$$\text{(T-Error)}\ \Gamma\ ;\ \texttt{true} \vdash \texttt{error} : T \qquad \text{(T-Try)}\ \frac{\Gamma\ ;\ \texttt{true} \vdash t_1 : T \quad \Gamma\ ;\ E \vdash t_2 : T}{\Gamma\ ;\ E \vdash \texttt{try}\ t_1\ \texttt{with}\ t_2 : T}$$

The remaining typing rules (those inherited from STLC), simply "push" the impurity predicate E unchanged from the conclusion into the premises. Intuitively, if a term is allowed to produce an error, all its sub-terms are also allowed to do so.

$$\text{(T-Var)}\ \frac{x : T \in \Gamma}{\Gamma\ ;\ E \vdash x : T} \qquad \text{(T-Abs)}\ \frac{\Gamma\ x : T_1\ ;\ E \vdash t_2 : T_2}{\Gamma\ ;\ E \vdash \lambda x : T_1.t_2 : T_1 \to T_2}$$

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

Progress of pure terms. We need to prove a more general statement first, namely a modified version of progress that states that closed, well-typed terms in normal form are either values or `error`s.

Lemma 1. *If $\emptyset\ ; E \vdash t : T$, then either (a) t is a value, (b) $t = \texttt{error}$, or (c) $t \longrightarrow t'$.*

Note that this lemma says nothing about the truth-value of the predicate E, it holds for both $E = \texttt{true}$ and $E = \texttt{true}$. In other words, the lemma should still hold for a simpler typing judgment $\Gamma \vdash t : T$, where we simply ignore E. This is exactly the typing judgment shown in Fig. 14-2 on p. 174 of TAPL, and the corresponding progress theorem carries over essentially unchanged.

Proof. The proof is by induction on typing derivations and most cases are essentially identical to those of the proof of Thm. 9.3.5, in TAPL (p. 105). We only show the cases for (T-App) and (T-Try) here.

For (T-App), we have $t = t_1\ t_2$, $\emptyset\ ; E \vdash t_1 : T_1 \to T_2$ and $\emptyset\ ; E \vdash t_2 : T_1$. By the IH, either t_1 is (a) a value, (b) an error, or (c) takes a step, and likewise for t_2. If t_1 can take a step, then (E-App1) applies to t; if t_1 is an error then (E-AppErr1) applies instead. If t_1 is a value, but t_2 can take a step or is an error, then either (E-App2) or (E-AppErr2) applies, respectively. Finally, if both t_1 and t_2 are values, then the canonical forms lemma tells us that $t_1 = \lambda x\colon T_1.t_3$ for some t_3, so that (E-AppAbs) applies to t.

For (T-Try), we have $t = \texttt{try}\ t_1\ \texttt{with}\ t_2$, $\emptyset\ ;\ \texttt{true} \vdash t_1 : T$ and $\emptyset\ ; E \vdash t_2 : T$, so by the IH, t_1 is either (a) a value, (b) an error, or (c) takes a step. In each case, we can take a step using, respectively, the rules (E-TryValue), (E-TryError) or (E-Try). □

With the above lemma, we can now proof the progress theorem for pure terms.

Proof. The proof is by induction on typing derivations and resembles again that of the progress theorem for the STLC (see TAPL, Thm. 9.3.5, p. 105).

The cases (T-Var), (T-Abs) and (T-App) are again identical to those in the proof for STLC (modulo the appearance of $E = \texttt{false}$ in the context) since the corresponding rules push the impurity predicate unchanged into the IH.

The case (T-Error) cannot occur since, by definition, its context is impure, i.e. $E = \texttt{true}$. The only remaining case is (T-Try). We have $t = \texttt{try}\ t_1\ \texttt{with}\ t_2$, $\emptyset\ ;\ \texttt{true} \vdash t_1 : T$ and $\emptyset\ ;\ \texttt{false} \vdash t_2 : T$. By the above lemma, t_1 is either (a) a value, (b) an error or (c) takes a step. Again, we can take a step in each case, using, respectively, the rules (E-TryValue), (E-TryError) or (E-Try). □

4 The call-by-value simply typed lambda calculus with returns

Consider a variant of the call-by-value simply typed lambda calculus specified in the appendix extended to support a new language construct: `return` t, which immediately returns a given term t from an **enclosing** lambda, disregarding any potential further computation typically needed for call-by-value evaluation rules.

The grammar of the extension is defined as follows. We distinguish top-level terms (tt) and nested terms (nt) to make sure that `return` t can only appear inside lambdas:

$$
\begin{array}{lcll}
v & ::= & \lambda x : \texttt{T} \,.\, nt \mid bv & (values) \\
bv & ::= & \texttt{true} \mid \texttt{false} & (boolean\ values) \\
nt & ::= & x \mid v \mid nt\ nt \mid \texttt{return}\ nt & (nested\ terms) \\
tt & ::= & x \mid v \mid tt\ tt & (top-level\ terms) \\
t & ::= & nt \mid tt & (terms) \\
p & ::= & tt & (programs) \\
T & ::= & \texttt{Bool} \mid \texttt{T} \to \texttt{T} & (types)
\end{array}
$$

In this exercise, you are to adjust the existing evaluation and typing rules, so that they correctly and comprehensively describe the semantics of the extension. More precisely, your task is two-fold:

1. Extend the evaluation rules (by adding new rules and/or changing existing ones) to express the early return semantics provided by return. Identify the evaluation strategy used by the specification and make sure that your extension adheres to it.

2. Extend the typing rules (by adding new rules and/or changing existing ones) to guarantee that types of values returned via return and via normal means are coherent. Make sure that progress and preservation conditions hold for your extension (you don't have to prove that formally, but your grade will be reduced if your extension ends up being unsafe).

   Hint: In addition to the immediate type of a term, you also need to keep track of the types of returned terms inside that term. For example, instead of the regular typing judgment $\Gamma \vdash t : \texttt{T}$, you can use the $\Gamma \vdash t : \texttt{T} \mid \texttt{R}$, where `R` is a set of types of terms, i.e. $\{T_1, ...T_n\}$, that can be returned from `t`.

Before you begin, think carefully about the following simple term: $\lambda x : \texttt{Bool}.\ (\texttt{return true})\ x$. Intuitively, it makes sense. Once this lambda is applied, it is going to evaluate to `true`, regardless of the input. Now, which typing rules would be used to type this term, so that it is accepted by our language? In particular, what type or types need to be assigned to `return true`?

4.1 Solution

Evaluation rules:

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

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

$$\frac{[x \to v_2]t_1 \longrightarrow^* \texttt{return}\ v_3}{(\lambda x : T.\ \ t_1)\ v_2 \longrightarrow v_3} \qquad \text{(E-AppAbs1)}$$

$$\frac{[x \to v_2]t_1 \longrightarrow^* v_3}{(\lambda x : T.\ t_1)\ v_2 \longrightarrow v_3} \quad \text{(E-AppAbs2)}$$

$$\frac{t \longrightarrow t'}{\texttt{return}\ t \longrightarrow \texttt{return}\ t'} \quad \text{(E-Ret)}$$

$$(\texttt{return}\ v_1)\ t_2 \longrightarrow \texttt{return}\ v_1 \quad \text{(E-AppRet1)}$$

$$t_1\ (\texttt{return}\ v_2) \longrightarrow \texttt{return}\ v_2 \quad \text{(E-AppRet2)}$$

$$\texttt{return}\ (\texttt{return}\ v) \longrightarrow \texttt{return}\ v \quad \text{(E-RetRet)}$$

Typing rules:

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T \mid \varnothing} \quad \text{(T-Var)}$$

$$\frac{\Gamma \vdash t : T \mid R}{\Gamma \vdash \texttt{return}\ t : T' \mid \{T\} \cup R} \quad \text{(T-Ret)}$$

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2 \mid \varnothing}{\Gamma \vdash (\lambda x : T_1.\ t_2) : T_1 \to T_2 \mid \varnothing} \quad \text{(T-Abs1)}$$

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2 \mid \{T_2\}}{\Gamma \vdash (\lambda x : T_1.\ t_2) : T_1 \to T_2 \mid \varnothing} \quad \text{(T-Abs2)}$$

$$\frac{\Gamma \vdash t_1 : T_2 \to T_1 \mid R_1 \quad \Gamma \vdash t_2 : T_2 \mid R_2}{\Gamma \vdash t_1\, t_2 : T_1 \mid R_1 \cup R_2} \quad \text{(T-App)}$$

$$\Gamma \vdash \texttt{true} : \texttt{Bool} \mid \varnothing \quad \text{(T-False)}$$

$$\Gamma \vdash \texttt{false} : \texttt{Bool} \mid \varnothing \quad \text{(T-True)}$$

4.1.1 Commentary

This is, of course, not the only possible solution. The typing rules presented treat R like their output, something that we can determine from the environment, term and the premise.

In (T-Var), the return set is always empty, because there's nothing else we can reasonably put there. In (T-Ret), we *extend* the current returns with the type of the value we're currently trying to return. In (T-App), we just take the union of the return sets of the two terms we have. This slightly comes back to bite us with the rule for typing abstractions, since we need two variants of the typing rule.

So do we need R to be a set in this task? We want R to sometimes be empty (because some expressions never explicitly `return`), and we want a convenient notion of "extending" R with another type. A set works quite well here. Try to think how you'd reformulate rules such as (T-App) without using sets.

5 Appendix

5.1 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:

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

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:

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

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

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

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

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