1. Maybe Types (64 pts.)

In many languages (e.g., C, Java) it is convenient to have a special “null” value that acts like a member of any reference type that is desired. However, the possibility that every reference may turn out to be null also creates difficulties for both the programmer and the language implementer. One way to have the expressive power of null without the undesirable side effects it to introduce a special type constructor \texttt{maybe} that effectively augments any type \( \tau \) with a special null value \( \langle \rangle \). Because the null value can be represented by a distinguished pointer value, a \texttt{maybe} \( \tau \) is easily implemented just as compactly as a C pointer or a Java reference. In this problem you will develop the semantics of maybes.

We start with the simply-typed \( \lambda \)-calculus with booleans \((\lambda \rightarrow)\) and extend it as follows:

- **Types**
  \[
  \tau ::= \ldots \mid \texttt{maybe} \ \tau
  \]

- **Terms**
  \[
  e ::= \ldots \mid \langle e \rangle \mid \langle \rangle \mid \texttt{if} \langle x \rangle = e_0 \texttt{then} e_1 \texttt{else} e_2
  \]

- **Values**
  \[
  v ::= \ldots \mid \langle v \rangle \mid \langle \rangle
  \]

Informally, the extensions work as follows. The new introduction form \( \langle e \rangle \) injects the value of \( e \) into the corresponding \texttt{maybe} type. The introduction form \( \langle \rangle \) is the special null value. The special \texttt{if} form checks whether an expression \( e_0 \) evaluates to a non-empty \texttt{maybe}; if so, the expression \( e_1 \) is evaluated with \( x \) bound to the injected value. If not, the expression \( e_2 \) is evaluated instead.

(a) (4 pts) Assuming left-to-right evaluation and the values given above, extend the small-step operational semantics in Pierce (Chp. 9) to maybes. Do not use evaluation contexts. Show only the new rules required to evaluate the maybe extensions shown above.

(b) (5 pts) Give any new typing rules that are required for the extended language.

(c) (20 pts) Extend the proofs of progress and preservation from \( \lambda \rightarrow \) as well as the proofs of any lemmas that the proofs of progress and preservation rely on—to demonstrate type soundness for this extended language \( \lambda \rightarrow \times \). Also, when proving preservation, use induction on the derivation of \( e \rightarrow e' \). The statements of the progress and preservation lemmas are as follows:

**Lemma (Progress):** If \( \vdash e : \tau \) then either \( e \) is a value or there exists some \( e' \) such that \( e \rightarrow e' \).

**Lemma (Preservation):** If \( \vdash e : \tau \) and \( e \rightarrow e' \), then \( \vdash e' : \tau \).

(d) (18 pts) Give a typed translation from this language \((\lambda \rightarrow \) extended with \texttt{maybe}\) to the language \( \lambda \rightarrow \texttt{Unit} \) (the simply-typed \( \lambda \)-calculus with sum types and type \texttt{Unit}). It should translate type derivations in the source language \((\lambda \rightarrow \texttt{maybe})\) to terms with type derivations in the target language \((\lambda \rightarrow \texttt{Unit})\), inductively demonstrating that any well-typed source term produces a well-typed target term.

Specifically, first define a translation function \( \mathcal{T}[\tau] \) that translates each source language type \( \tau \) to a target language type. Next, define a type-preserving translation function \( \mathcal{E}[\cdot] \) that, when applied to a source language typing judgment, produces a well-typed target term. It will be useful to have a function \( \mathcal{G}[\cdot] \) that
2. **Subtyping** (16 pts.)

For each of the following questions, answer Yes or No. If the answer is Yes, show the subtyping derivation. If the answer is No, give either a term that demonstrates how type safety breaks if we allow the two types in the subtype relation, or a short explanation of why type safety is preserved even if we allow the two types in the subtype relation.

(a) (4 pts) Is \{x : \text{Top} \rightarrow \text{Ref} \} a subtype of \{x : \text{Top} \rightarrow \text{Top} \}?
(b) (4 pts) Is \{x : \text{Top} \rightarrow \text{Ref} \} a subtype of \{x : \text{Ref} \rightarrow \text{Ref} \{y : \text{Top}\}\}?
(c) (4 pts) Is \{x : \text{Ref} \{y : \text{Top}\}\} a subtype of \{x : \text{Ref} \}\?
(d) (4 pts) Is \{x : \text{Top}\} a subtype of \{x : \{\}\}\?

3. **Strong normalization** (20 pts.)

Let us add tagged sums to the simply-typed \(\lambda\)-calculus \((\lambda^+)^+\). We’ll denote this calculus \(\lambda^{++}\).

| Types | \(\tau \ ::= \ldots \mid \tau_1 + \tau_2\) |
| Terms | \(e \ ::= \ldots \mid \text{inl}_{\tau_1+\tau_2} e \mid \text{inr}_{\tau_1+\tau_2} e \mid \text{case } e \text{ of inl } x \Rightarrow e_1 \mid \text{inr } y \Rightarrow e_2\) |
| Values | \(v \ ::= \ldots \mid \text{inl}_{\tau_1+\tau_2} v \mid \text{inr}_{\tau_1+\tau_2} v\) |
| Eval. Contexts | \(E \ ::= \ldots \mid \text{inl}_{\tau_1+\tau_2} E \mid \text{inr}_{\tau_1+\tau_2} E \mid \text{case } E \text{ of inl } x \Rightarrow e_1 \mid \text{inr } y \Rightarrow e_2\) |

**New reduction rules:**

- \(\text{case } \text{inl}_{\tau_1+\tau_2} v \text{ of inl } x \Rightarrow e_1 \mid \text{inr } y \Rightarrow e_2 \Rightarrow e_1[v/x]\) (E-CaseINL)
- \(\text{case } \text{inr}_{\tau_1+\tau_2} v \text{ of inl } x \Rightarrow e_1 \mid \text{inr } y \Rightarrow e_2 \Rightarrow e_2[v/y]\) (E-CaseINR)

**New typing rules:**

\[
\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \text{inl}_{\tau_1+\tau_2} e : \tau_1 + \tau_2} \quad \text{(T-INL)}
\]
\[
\frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \text{inr}_{\tau_1+\tau_2} e : \tau_1 + \tau_2} \quad \text{(T-INR)}
\]
\[
\frac{\Gamma \vdash e : \tau_1 + \tau_2 \quad \Gamma, x : \tau_1 \vdash e_1 : \tau \quad \Gamma, y : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \text{case } e \text{ of inl } x \Rightarrow e_1 \mid \text{inr } y \Rightarrow e_2 : \tau} \quad \text{(T-CASE)}
\]

Show that all expressions in the language \(\lambda^{++}\) are strongly normalizing by extending the proof of strong normalization for \(\lambda^+\).