You may turn in handwritten solutions to this assignment—but make sure you write clearly and leave lots of whitespace. If you choose to typeset your solutions, the LaTeX sources are available for your use at the course website. (Look in cs7480.sty for macros you can use.) A pdf file containing the solutions can be submitted via email.

This homework is worth 100 points.

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 **maybe** that effectively augments any type τ with a special null value $\langle\rangle$. Because the null value can be represented by a distinguished pointer value, a **maybe** τ 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 λ-calculus with booleans ($(\lambda^{\rightarrow}$) and extend it as follows:

   $$\begin{array}{llllll} \textit{Types} & \tau & ::= & \ldots & \mid & \textbf{maybe}\ \tau \\ \textit{Terms} & e & ::= & \ldots \mid \langle e\rangle \mid \langle\rangle & \mid & \textbf{if}\ \langle x\rangle = e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \\ \textit{Values} & v & ::= & \ldots \mid \langle v\rangle \mid \langle\rangle & & \end{array}$$

   Informally, the extensions work as follows. The new introduction form $\langle e\rangle$ injects the value of e into the corresponding **maybe** type. The introduction form $\langle\rangle$ is the special null value. The special **if** form checks whether an expression e_0 evaluates to a non-empty 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 $e \longrightarrow e'$. 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 $e \longrightarrow e'$.

   Lemma (Preservation): If $\vdash e : \tau$ and $e \longrightarrow e'$, then $\vdash e' : \tau$.

   (d) (18 pts) Give a typed translation from this language ($\lambda^{\rightarrow}$ extended with **maybe**) to the language $\lambda^{\rightarrow+\mathbf{Unit}}$ (the simply-typed λ-calculus with sum types and type **Unit**). It should translate type derivations in the source language ($\lambda^{\rightarrow\ \mathbf{maybe}}$) to terms with type derivations in the target language ($\lambda^{\rightarrow+\mathbf{Unit}}$) , 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 τ 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

simply maps the types of all the variables in Γ into the target language:

$$\begin{array}{lcl} \mathcal{G}[\![\emptyset]\!] & = & \emptyset \\ \mathcal{G}[\![\Gamma, x{:}\tau]\!] & = & \mathcal{G}[\![\Gamma]\!], x{:}\mathcal{T}[\![\tau]\!] \end{array}$$

Now, define $\mathcal{E}[\![\cdot]\!]$ in such a way that if $\Gamma \vdash e : \tau$ in the source language, then in the target language, we should have:

$$\mathcal{G}[\![\Gamma]\!] \vdash \mathcal{E}[\![\Gamma \vdash e : \tau]\!] : \mathcal{T}[\![\tau]\!].$$

(e) (8 pts) Define the weakest sound subtyping relationship on types **maybe** τ and **maybe** τ' and justify it by defining the appropriate coercion function.

(f) (5 pts) Do the same for **maybe** τ and τ. Why would such a subtype relationship be helpful?

(g) (4 pts) Given the syntax of the language and the typing rules that you gave in part (b) above, will every well-typed term in this language have a unique type? That is, does the Uniqueness of Types theorem (Pierce, Theorem 9.3.3) hold? If so, briefly explain why that must be the case; if not, briefly say why not, and give the minimal changes necessary to ensure uniqueness of types.

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 $\{\mathsf{x} : \mathsf{Top} \to \mathsf{Ref\ Top}\}$ a subtype of $\{\mathsf{x} : \mathsf{Top} \to \mathsf{Top}\}$?

(b) (4 pts) Is $\{\mathsf{x} : \mathsf{Top} \to \mathsf{Ref\ Top}\}$ a subtype of $\{\mathsf{x} : \mathsf{Ref\ Top} \to \mathsf{Ref}\ \{\mathsf{y} : \mathsf{Top}\}\}$?

(c) (4 pts) Is $\{\mathsf{x} : \mathsf{Ref}\ \{\mathsf{y} : \mathsf{Top}\}\}$ a subtype of $\{\mathsf{x} : \mathsf{Ref\ Top}\}$?

(d) (4 pts) Is $\{\mathsf{x} : \mathsf{Top}\}$ a subtype of $\{\mathsf{x} : \{\,\}\}$?

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

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

Types	τ	::=	$\ldots \mid \tau_1 + \tau_2$
Terms	e	::=	$\ldots \mid \mathbf{inl}_{\tau_1+\tau_2}\, e \mid \mathbf{inr}_{\tau_1+\tau_2}\, e \mid \mathbf{case}\ e\ \mathbf{of\ inl}\ x \Rightarrow e_1 \mid \mathbf{inr}\ y \Rightarrow e_2$
Values	v	::=	$\ldots \mid \mathbf{inl}_{\tau_1+\tau_2}\, v \mid \mathbf{inr}_{\tau_1+\tau_2}\, v$
Eval. Contexts	E	::=	$\ldots \mid \mathbf{inl}_{\tau_1+\tau_2}\, E \mid \mathbf{inr}_{\tau_1+\tau_2}\, E \mid \mathbf{case}\ E\ \mathbf{of\ inl}\ x \Rightarrow e_1 \mid \mathbf{inr}\ y \Rightarrow e_2$

New reduction rules:

$$\mathbf{case\ inl}_{\tau_1+\tau_2}\, v\ \mathbf{of\ inl}\ x \Rightarrow e_1 \mid \mathbf{inr}\ y \Rightarrow e_2 \longrightarrow e_1[v/x] \qquad \text{(E-CaseInl)}$$

$$\mathbf{case\ inr}_{\tau_1+\tau_2}\, v\ \mathbf{of\ inl}\ x \Rightarrow e_1 \mid \mathbf{inr}\ y \Rightarrow e_2 \longrightarrow e_2[v/y] \qquad \text{(E-CaseInr)}$$

New typing rules:

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \mathbf{inl}_{\tau_1+\tau_2}\, e : \tau_1 + \tau_2}\ \text{(T-Inl)} \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \mathbf{inr}_{\tau_1+\tau_2}\, e : \tau_1 + \tau_2}\ \text{(T-Inr)}$$

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2 \qquad \Gamma, x : \tau_1 \vdash e_1 : \tau \qquad \Gamma, y : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \mathbf{case}\ e\ \mathbf{of\ inl}\ x \Rightarrow e_1 \mid \mathbf{inr}\ y \Rightarrow e_2 : \tau}\ \text{(T-Case)}$$

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