You can turn in handwritten solutions to this assignment. To keep your graders happy, please write clearly, leave lots of whitespace, and use standard-sized (8.5 by 11in) paper! You may be penalized upto 20 points if you do not follow these instructions. Handwritten solutions should be submitted at Lindley Hall 301G by 5pm on the due date.

If you choose to typeset your solutions, you may use LaTeX or Word. If you use LaTeX, there is a template available for your use at the course website. (Remember to look in b522.sty for macros you can use.) A pdf file containing the solutions can be submitted online by midnight on the due date.

This homework is worth 100 points. Problems 2 and 3 will require some thought, so start early!

1. **Type Soundness Warmup** (20 pts.)

   We saw that the simply-typed λ-calculus has a sound type system because it preserves types and guarantees progress of well-typed terms. Thus, well-typed terms do not get stuck (i.e., evaluation is *safe*). Let us add pair terms and product types to the simply-typed λ-calculus ($\lambda^{\rightarrow}$).

$$\begin{array}{llll} \textit{Types} & T & ::= & \ldots \mid T_1 \times T_2 \\ \textit{Terms} & e & ::= & \ldots \mid (e_1, e_2) \mid \textbf{fst}\ e \mid \textbf{snd}\ e \\ \textit{Values} & v & ::= & \ldots \mid (v_1, v_2) \end{array}$$

   New evaluation rules:

$$\frac{e_1 \longrightarrow e_1'}{(e_1, e_2) \longrightarrow (e_1', e_2)}\ \text{(E-Pair1)} \qquad \frac{e_2 \longrightarrow e_2'}{(v_1, e_2) \longrightarrow (v_1, e_2')}\ \text{(E-Pair2)}$$

$$\frac{e \longrightarrow e'}{\textbf{fst}\ e \longrightarrow \textbf{fst}\ e'}\ \text{(E-Fst)} \qquad \frac{e \longrightarrow e'}{\textbf{snd}\ e \longrightarrow \textbf{snd}\ e'}\ \text{(E-Snd)}$$

$$\frac{}{\textbf{fst}\ (v_1, v_2) \longrightarrow v_1}\ \text{(E-FstPair)} \qquad \frac{}{\textbf{snd}\ (v_1, v_2) \longrightarrow v_2}\ \text{(E-SndPair)}$$

   New typing rules:

$$\frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma \vdash e_2 : T_2}{\Gamma \vdash (e_1, e_2) : T_1 \times T_2}\ \text{(T-Pair)}$$

$$\frac{\Gamma \vdash e : T_1 \times T_2}{\Gamma \vdash \textbf{fst}\ e : T_1}\ \text{(T-Fst)} \qquad \frac{\Gamma \vdash e : T_1 \times T_2}{\Gamma \vdash \textbf{snd}\ e : T_2}\ \text{(T-Snd)}$$

   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 $\vdash e : T$ then *either* e is a value *or* there exists some e' such that $e \longrightarrow e'$.

   Lemma (Preservation): If $\vdash e : T$ and $e \longrightarrow e'$, then $\vdash e' : T$.

2. **Explicit Initialization** (35 pts.)

Compound data structures, e.g., arrays, tuples, and records, often need to be initialized step by step, rather than being created atomically.

For example, in the Java language, when an object is created, before the execution of its constructor, all the non-primitive-typed fields have the default value **null**. The object is then gradually initialized using individual assignments to the fields.

Now let us try to model step-by-step initialization of tuples in the context of the simply-typed λ-calculus:

$$
\begin{array}{llll}
\textit{Ground values} & b & ::= & \textbf{null} \mid \textbf{true} \mid \textbf{false} \mid n \\
\textit{Values} & v & ::= & b \mid \lambda x{:}T.\, e \mid (v_1, \ldots, v_n) \\
\textit{Terms} & e & ::= & v \mid x \mid e_1\ e_2 \mid \textbf{malloc}\ T_1 \times \ldots \times T_n \mid \#i\ e \mid \#i\ e_1 := e_2 \\
 & & & \\
\textit{Ground types} & B & ::= & \textbf{Unit} \mid \textbf{Bool} \mid \textbf{Int} \\
\textit{Types} & T & ::= & B \mid T_1 \to T_2 \mid (T_1 \times \ldots \times T_n)\backslash\{i_1, \ldots, i_k\}
\end{array}
$$

In order to create a tuple, the expression $\textbf{malloc}\ T_1 \times \ldots \times T_n$ is used, rather than $(e_1, \ldots, e_n)$ which—as we saw in class—creates a fully initialized tuple at once. The result of $\textbf{malloc}\ T_1 \times \ldots \times T_n$ is a fully *uninitialized* tuple, $(\textbf{null}, \ldots, \textbf{null})$, of type $(T_1 \times \ldots \times T_n)\backslash\{1, \ldots, n\}$.

The type $(T_1 \times \ldots \times T_n)\backslash\{i_1, \ldots, i_k\}$ is called a *masked type*, which represents a tuple that has not been fully initialized: the elements numbered $i_1, \ldots, i_k$ are *masked*— that is, they are not initialized and have the value **null**. The tuple, after being fully initialized, should have the type $T_1 \times \ldots \times T_n$, which we assume is syntactic sugar for the type $(T_1 \times \ldots \times T_n)\backslash\{\}$.

Tuples are initialized *functionally* with expressions $\#i\ e_1 := e_2$, in which e_1 first evaluates to a tuple with its i-th element masked, and e_2 evaluates to a value that is compatible with the type of the i-th element in the tuple. The expression will generate a new tuple with its i-th element initialized, and otherwise the same as the result of e_1. Note that each element of a tuple should only be initialized once.

To project the i-th element of a tuple, the expression $\#i\ e$ is used. Note, however, that projection of uninitialized elements is prohibited.

For example, the following expression will evaluate to a tuple $(10, 20)$ of type $\textbf{Int} \times \textbf{Int}$.

$$
\begin{array}{l}
(\lambda x{:}(\textbf{Int} \times \textbf{Int})\backslash\{2\}.\ \#2\ x := 20) \\
\quad ((\lambda x{:}(\textbf{Int} \times \textbf{Int})\backslash\{1, 2\}.\ \#1\ x := 10) \\
\quad\quad (\textbf{malloc Int} \times \textbf{Int}))
\end{array}
$$

(a) (7 pts) Extend the small-step operational semantics of the simply-typed λ-calculus to include the new expressions: $(v_1, \ldots, v_n)$, $\textbf{malloc}\ T_1 \times \ldots \times T_n$, $\#i\ e$, and $\#i\ e_1 := e_2$. Specifically, assuming left-to-right evaluation, extend the definition of the evaluation contexts and give the additional reduction rules required.

(b) (10 pts) Extend the typing rules of the simply-typed λ-calculus to include the new constructs.

(c) (18 pts) Prove the soundness of the type system. (For each of the lemmas involved, you only need to show the proofs for cases that involve the new constructs.)

3. **Maybe Types** (45 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. A neat 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 T with a special null value $\langle\rangle$. Because the null value can be represented by a distinguished pointer value, a **maybe** T 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 and extend it as follows:

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

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) (5 pts) Assuming left-to-right evaluation and the values given above, define how to extend the legal evaluation contexts E in which reductions can occur. Also, give rules defining the new reductions needed for the extended language.

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

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

Specifically, first define a translation function $\mathcal{T}[\![T]\!]$ that translates each source language type T 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:

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

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:

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

(d) (8 pts) Define the weakest sound subtyping relationship on types **maybe** T and **maybe** T' and justify it by defining the appropriate coercion function.

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

(f) (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.