

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, there is a LaTeX template available for your use at the course website. (Remember to look in `cs7480.sty` for macros you can use.) A pdf file containing the solutions can be submitted via email.

This homework is worth 150 points.

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 call-by-value (CBV) simply-typed λ -calculus (λ^{\rightarrow}).

<i>Types</i>	$\tau ::= \dots$		$\tau_1 \times \tau_2$
<i>Terms</i>	$e ::= \dots$		$(e_1, e_2) \mid \mathbf{fst} \ e \mid \mathbf{snd} \ e$
<i>Values</i>	$v ::= \dots$		(v_1, v_2)
<i>Eval. Contexts</i>	$E ::= \dots$		$(E, e_2) \mid (v_1, E) \mid \mathbf{fst} \ E \mid \mathbf{snd} \ E$

New reduction rules:

$$\begin{aligned} \mathbf{fst} \ (v_1, v_2) &\longrightarrow v_1 && \text{(E-FSTPAIR)} \\ \mathbf{snd} \ (v_1, v_2) &\longrightarrow v_2 && \text{(E-SNDPAIR)} \end{aligned}$$

New typing rules:

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \quad \text{(T-PAIR)}$$

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \mathbf{fst} \ e : \tau_1} \quad \text{(T-FST)} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \mathbf{snd} \ e : \tau_2} \quad \text{(T-SND)}$$

Extend the proofs of progress and preservation from λ^{\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 $\vdash e' : \tau$.

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, when an object is created in Java, before the execution of its constructor, all the non-primitive-typed fields have the default value (). 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:

<i>Ground values</i>	$b ::= () \mid \mathbf{true} \mid \mathbf{false} \mid n$
<i>Values</i>	$v ::= b \mid \lambda x:\tau. e \mid (v_1, \dots, v_n)$
<i>Terms</i>	$e ::= v \mid x \mid e_1 e_2 \mid \mathbf{malloc} \tau_1 \times \dots \times \tau_n \mid \#i e \mid \#i e_1 := e_2$
<i>Ground types</i>	$B ::= \mathbf{Unit} \mid \mathbf{Bool} \mid \mathbf{Int}$
<i>Types</i>	$\tau ::= B \mid \tau_1 \rightarrow \tau_2 \mid (\tau_1 \times \dots \times \tau_n) \setminus \{i_1, \dots, i_k\}$

In order to create a tuple, the expression $\mathbf{malloc} \tau_1 \times \dots \times \tau_n$ is used, rather than (e_1, \dots, e_n) which—as we saw in class—creates a fully initialized tuple at once. The result of $\mathbf{malloc} \tau_1 \times \dots \times \tau_n$ is a fully *uninitialized* tuple, $((), \dots, ())$, of type $(\tau_1 \times \dots \times \tau_n) \setminus \{1, \dots, n\}$.

The type $(\tau_1 \times \dots \times \tau_n) \setminus \{i_1, \dots, i_k\}$ is called a *masked type*, which represents a tuple that has not been fully initialized: the elements numbered i_1, \dots, i_k are *masked*—that is, they are not initialized and have the value (). The tuple, after being fully initialized, should have the type $\tau_1 \times \dots \times \tau_n$, which we assume is syntactic sugar for the type $(\tau_1 \times \dots \times \tau_n) \setminus \{\}$.

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 $\mathbf{Int} \times \mathbf{Int}$.

$$\begin{aligned} & (\lambda x:(\mathbf{Int} \times \mathbf{Int}) \setminus \{2\}. \#2 x := 20) \\ & ((\lambda x:(\mathbf{Int} \times \mathbf{Int}) \setminus \{1, 2\}. \#1 x := 10) \\ & \quad (\mathbf{malloc} \mathbf{Int} \times \mathbf{Int})) \end{aligned}$$

- (7 pts) Extend the small-step operational semantics of the simply-typed λ -calculus to include the new expressions: (v_1, \dots, v_n) , $\mathbf{malloc} \tau_1 \times \dots \times \tau_n$, $\#i e$, 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.
- (10 pts) Extend the typing rules of the simply-typed λ -calculus to include the new constructs.
- (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 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 and extend it as follows:

$$\begin{array}{lcl} \text{Types} & \tau & ::= \dots \mid \mathbf{maybe} \tau \\ \text{Terms} & e & ::= \dots \mid \langle e \rangle \mid \langle \rangle \mid \mathbf{if} \langle x \rangle = e_0 \mathbf{then} e_1 \mathbf{else} e_2 \\ \text{Values} & v & ::= \dots \mid \langle v \rangle \mid \langle \rangle \end{array}$$

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) (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 (λ^{\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:

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

- (d) (8 pts) Define the weakest sound subtyping relationship on types **maybe** τ and **maybe** τ' and justify it by defining the appropriate coercion function.
- (e) (5 pts) Do the same for **maybe** τ and τ . 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.

4. **Recursive types** (10 pts.)

Consider mutually recursive type definitions like the following:

```
type Node = Edge list
type Edge = Node * Node
```

Eliminate the mutual recursion by giving recursive (μ) types for **Node** and **Edge**, and show that the unfoldings of your **Node** and **Edge** types satisfy their respective equations. You may assume that **list** and \times (which is the notation we've been using for the $*$ type) are built-in type constructors.

5. **Subtyping** (20 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) (5 pts) Is $\{x : \text{Top} \rightarrow \text{Ref Top}\}$ a subtype of $\{x : \text{Top} \rightarrow \text{Top}\}$?
- (b) (5 pts) Is $\{x : \text{Top} \rightarrow \text{Ref Top}\}$ a subtype of $\{x : \text{Ref Top} \rightarrow \text{Ref } \{y : \text{Top}\}\}$?
- (c) (5 pts) Is $\{x : \text{Ref } \{y : \text{Top}\}\}$ a subtype of $\{x : \text{Ref Top}\}$?
- (d) (5 pts) Is $\{x : \text{Top}\}$ a subtype of $\{x : \{\}\}$?

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

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

<i>Types</i>	$\tau ::= \dots \mid \tau_1 + \tau_2$
<i>Terms</i>	$e ::= \dots \mid \mathbf{inl}_{\tau_1+\tau_2} e \mid \mathbf{inr}_{\tau_1+\tau_2} e \mid \mathbf{case } e \mathbf{ of inl } x \Rightarrow e_1 \mid \mathbf{inr } y \Rightarrow e_2$
<i>Values</i>	$v ::= \dots \mid \mathbf{inl}_{\tau_1+\tau_2} v \mid \mathbf{inr}_{\tau_1+\tau_2} v$
<i>Eval. Contexts</i>	$E ::= \dots \mid \mathbf{inl}_{\tau_1+\tau_2} E \mid \mathbf{inr}_{\tau_1+\tau_2} E \mid \mathbf{case } E \mathbf{ of inl } x \Rightarrow e_1 \mid \mathbf{inr } y \Rightarrow e_2$

New reduction rules:

$$\begin{aligned} \mathbf{case } \mathbf{inl}_{\tau_1+\tau_2} v \mathbf{ of inl } x \Rightarrow e_1 \mid \mathbf{inr } y \Rightarrow e_2 &\longrightarrow e_1[v/x] && \text{(E-CASEINL)} \\ \mathbf{case } \mathbf{inr}_{\tau_1+\tau_2} v \mathbf{ of inl } x \Rightarrow e_1 \mid \mathbf{inr } y \Rightarrow e_2 &\longrightarrow e_2[v/y] && \text{(E-CASEINR)} \end{aligned}$$

New typing rules:

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

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