1 Coercion semantics [25 points]

In this question, we study the coercion semantics for subtyping. Under the coercion semantics, a subtyping relation \( A \leq B \) holds if there exists a method to convert values of type \( A \) to values of type \( B \). As a witness to the existence of such a method, we usually use a \( \lambda \)-abstraction, called a coercion function, of type \( A \to B \). We use a coercion subtyping judgment

\[
A \leq B \Rightarrow f
\]

to mean that \( A \leq B \) holds under the coercion semantics with a coercion function \( f \) of type \( A \to B \). For example, a judgment \( \text{int} \leq \text{float} \Rightarrow \text{int2float} \) holds if the coercion function \( \text{int2float} \) converts integers of type \( \text{int} \) to floating point numbers of type \( \text{float} \).

The following is a subtyping system for the coercion semantics for the simply-typed \( \lambda \)-calculus. The rules \( \text{Refl}_\leq \) and \( \text{Trans}_\leq \) express reflexivity and transitivity of the subtyping relation, respectively. Define the subtyping rules for product types and function types:

\[
\frac{}{A \leq A} \quad \frac{A \leq B \Rightarrow f \quad B \leq C \Rightarrow g}{A \leq C \Rightarrow \lambda x : A.g (f x)} \quad \frac{A \leq A' \Rightarrow f \quad B \leq B' \Rightarrow g}{A \times B \leq A' \times B' \Rightarrow \lambda x : A \times B.(f \ (\text{fst} \ x), g \ (\text{snd} \ x))} \quad \frac{A' \leq A \Rightarrow f \quad B \leq B' \Rightarrow g}{A \to B \leq A' \to B' \Rightarrow \lambda h : A \to B. \lambda x : A'.g (h \ (f x))}
\]

2 Recursion with recursive types [75 points]

Consider the (call-by-value) simply typed \( \lambda \)-calculus with recursive types, which we refer to as \( L_1 \). For the sake of simplicity, we do not annotate fold and unfold with a recursive type:

\[
L_1 = \left\{ \begin{array}{ll}
\text{type} & A ::= A \to A \mid \alpha \mid \mu \alpha.A \\
\text{expression} & e ::= x \mid \lambda x : A.e \mid e \circ e \mid \text{fold } e \mid \text{unfold } e \\
\text{value} & v ::= \lambda x : A.e \mid \text{fold } v
\end{array} \right.
\]

It turns out that \( L_1 \) permits recursive functions even without the fixed point construct. The idea is that we translate the fixed point combinator fix of the untyped \( \lambda \)-calculus to \( \text{fix}^\circ \) of \( L_1 \). Unfortunately \( \text{fix}^\circ \) is not particularly useful because it has type \( \Omega = \alpha \to \alpha \). For example, we cannot use \( \text{fix}^\circ \) to directly implement a recursive function on integers (such as factorial function, Fibonacci function, and so on) as we would in SML.

In this problem, we will investigate how to directly implement recursive functions in \( L_1 \) using recursive types. We first extend \( L_1 \) with a new form of type \( A \Rightarrow B \) and two new forms of expressions \( \text{fun } x : A.e \) and \( e \circ e \); we refer to the resultant language as \( L_2 \):

\[
L_2 = L_1 + \left\{ \begin{array}{ll}
\text{type} & A ::= \cdots \mid A \Rightarrow A \\
\text{expression} & e ::= \cdots \mid \text{fun } f : A.e \mid e \circ e \\
\text{value} & v ::= \cdots \mid \text{fun } f : A.e
\end{array} \right.
\]

A recursive function construct \( \text{fun } f : x : A.e \) defines a recursive function \( f \) whose argument is \( x \) of type \( A \) and whose body is \( e \). A recursive function application \( e_1 \circ e_2 \) applies a recursive function obtained by evaluating \( e_1 \) to the result of evaluating \( e_2 \). We use a recursive function type \( A \Rightarrow B \) for recursive functions from type \( A \) to type \( B \):

\[
\frac{\Gamma, f : A \Rightarrow B, x : A \vdash e : B}{\Gamma \vdash \text{fun } f : x : A.e : A \Rightarrow B} \quad \frac{\Gamma \vdash e : A \Rightarrow B}{\Gamma \vdash e \circ e' : A \Rightarrow B} \quad \frac{\Gamma \vdash e_1 : A \Rightarrow B \quad \Gamma \vdash e_2 : B}{\Gamma \vdash (\text{fun } f : A.e) \circ e_2 \Rightarrow (\text{fun } f : A.e) \circ e_2} \quad \text{RArg}
\]
\[(\text{fun } f \; x: A. \; e) \otimes v \mapsto [\text{fun } f \; x: A. e/[v/x]e]^RApp\]

The goal is to show that \(L_2\) is equivalent to \(L_1\): adding \(\text{fun } f \; x: A. \; e\) and \(e \otimes e\) to \(L_1\) does not actually increase its expressive power. To be precise, you will show that:

- \(A \Rightarrow B\) can be defined in terms of ordinary function types and recursive types in \(L_1\).
- \(\text{fun } f \; x: A. \; e\) and \(e \otimes e\) can be defined in terms of those constructs in \(L_1\).

**Question 1. [25 points]** Complete the definition of a translation function \((\cdot)^*\) which translates a type \(A\) in \(L_2\) to a type \(A^*\) in \(L_1\).

HINT: The key idea in the translation is essentially the same as in developing the fixed point combinator \(\text{fix}\) for the untyped \(\lambda\)-calculus: self-application. Informally speaking, a recursive function takes itself as its first argument. Perhaps the rules \(\Rightarrow I\) and \(RApp\) are the best hint that we could give.

Fill in the blank:

\[
(A \Rightarrow B)^* = A^* \Rightarrow B^* \\
(\alpha^*) = \alpha \\
((\mu \alpha. A)^*) = \mu \alpha. A^*
\]

**Question 2. [25 points]** Complete the definition of a translation function \((\cdot)^*\) which translates an expression \(e\) in \(L_2\) to an expression \(e^*\) in \(L_1\). The translation function must satisfy the following invariant:

**Invariant.** If \(\Gamma \vdash e : A\) holds in \(L_2\), then \(\Gamma^* \vdash e^* : A^*\) holds in \(L_1\), where \(\Gamma^*\) converts each type binding \(x : A\) in \(\Gamma\) to \(x : A^*\).

Fill in the blank below. For the case \((\text{fun } f \; x: A. \; e)^*\), we assume that \(\text{fun } f \; x: A. \; e\) has type \(A \Rightarrow B\) in \(L_2\).

\[
x^* = x \\
(\lambda x: A. e)^* = \lambda x: A^*. e^* \\
(e_1 e_2)^* = e_1^* e_2^* \\
(\text{fold } e)^* = \text{fold } e^* \\
(\text{unfold } e)^* = \text{unfold } e^* \\
(fold e)^* = \text{fold } e^* \\
(\text{unfold } e)^* = \text{unfold } e^*
\]

\[
(\text{fun } f \; x: A. e)^* = \text{fold } \lambda f : (A \Rightarrow B)^*. \lambda x: A^*. e^* \\
(e_1 \otimes e_2)^* = \text{unfold } e_1^* e_2^*
\]

**Question 3. [25 points]** Show the reduction sequence of \(((\text{fun } f \; x: A. e) \otimes v)^*\). We assume that \(\text{fun } f \; x: A. e\) has type \(A \Rightarrow B\) in \(L_2\). We also assume that no variable capture occurs in substitutions during the reduction.

\[
((\text{fun } f \; x: A. e) \otimes v)^* = \text{unfold } (\text{fold } \lambda f : (A \Rightarrow B)^*. \lambda x: A^*. e^*) (\text{fun } f \; x: A. e)^* v^* \\
\Rightarrow (\lambda f : (A \Rightarrow B)^*. \lambda x: A^*. e^*) (\text{fun } f \; x: A. e)^* v^* \\
\Rightarrow (\lambda x: A^*. [\text{fun } f \; x: A. e]^*/f[e^*] v^* \\
\Rightarrow [v^*/x][\text{fun } f \; x: A. e]^*/f[e^*]
\]