1 Type reconstruction for System F [15 points]

The abstract syntax for System F is as follows:

\[
\begin{align*}
\text{type} & : \quad A ::= A \to A \mid \alpha \mid \forall \alpha.A \\
\text{expression} & : \quad e ::= x \mid \lambda x : A.e \mid e \; e \mid \Lambda \alpha.e \mid e[A]
\end{align*}
\]

In the case of System F, the goal of type reconstruction is to convert an expression \( e \) in the untyped \( \lambda \)-calculus to a well-typed expression \( e' \) in System F. Explain the relationship between such a pair of expressions \( e \) (in the untyped \( \lambda \)-calculus) and \( e' \) (in System F).

Answer: Erasing type annotations (including type abstractions and type applications) in \( e' \) yields the original expression \( e \). The erasure function is defined as follows:

\[
\begin{align*}
\text{erase}(x) & = x \\
\text{erase}(\lambda x : A.e) & = \lambda x. \text{erase}(e) \\
\text{erase}(e_1 e_2) & = \text{erase}(e_1) \text{erase}(e_2) \\
\text{erase}(\Lambda \alpha.e) & = \text{erase}(e) \\
\text{erase}(e[A]) & = \text{erase}(e)
\end{align*}
\]

2 Let-polymorphism [35 points]

Question 1. [20 points] In the let-polymorphic \( \lambda \)-calculus, what is the problem with allowing a \( \lambda \)-abstraction \( \lambda x : U.e \) binding \( x \) to a polytype (which would degenerate \( \text{let } x : U = e \text{ in } e' \) into syntactic sugar)?

Answer: The reason is that with an additional assumption that \( e \) may have a polytype (e.g., \( \lambda x : U.x \)), such a \( \lambda \)-abstraction collapses the distinction between monotypes and polytypes. That is, polytypes constitute types of System F:

\[
\begin{align*}
\text{monotype} & \quad A ::= U \to U \mid \alpha \\
\text{polytype} & \quad U ::= A \mid \forall \alpha.U \\&
\end{align*}
\]

Question 2. [15 points] Explain the key idea behind let-polymorphism.

Answer: Use a let-binding as a restricted use of a \( \lambda \)-abstraction \( \lambda x : U.e \) (binding \( x \) to a polytype) such that it never stands alone as a first-class object and must be applied to a polymorphic expression immediately.

3 Implicit polymorphism [50 points]

Consider the implicit let-polymorphic \( \lambda \)-calculus:

\[
\begin{align*}
\text{monotype} & \quad A ::= A \to A \mid \alpha \\
\text{polytype} & \quad U ::= A \mid \forall \alpha.U \\
\text{expression} & \quad e ::= x \mid \lambda x.e \mid e \; e \mid \text{let } x = e \text{ in } e \\
\text{value} & \quad v ::= \lambda x.e \\
\text{typing context} & \quad \Gamma ::= \cdot \mid \Gamma, x : U \mid \Gamma, \alpha \text{ type}
\end{align*}
\]

We use a typing judgment \( \Gamma \vdash e : U \) to express that untyped expression \( e \) is typable with a polytype \( U \).

Question 3. [30 points] Complete the typing rules for the implicit let-polymorphic \( \lambda \)-calculus.

\[
\begin{align*}
\Gamma \vdash x : U & \quad \text{Var} \\
\Gamma, x : A \vdash e : B & \quad \Gamma \vdash \lambda x.e : A \to B \\
\Gamma \vdash &
\end{align*}
\]
Question 4. [20 points] Complete the typing derivation for expression \( \lambda x. \lambda y. (x, y) \) in the implicit let-polymorphic type system. For typing pairs \( (x, y) \), you may assume product types \( A \times B \) and their typing rules.