CSE-321 Programming Languages 2012
Final — Sample Solution

<table>
<thead>
<tr>
<th>Prob 1</th>
<th>Prob 2</th>
<th>Prob 3</th>
<th>Prob 4</th>
<th>Prob 5</th>
<th>Prob 6</th>
<th>Prob 7</th>
<th>Prob 8</th>
<th>Prob 9</th>
<th>Prob 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>11</td>
<td>11</td>
<td>10</td>
<td>7</td>
<td>17</td>
<td>4</td>
<td>8</td>
<td>6</td>
<td>16</td>
</tr>
</tbody>
</table>

- There are ten problems on 20 pages in this exam.
- The maximum score for this exam is 100 points.
- Be sure to write your name and Hemos ID.
- You have three hours for this exam.
1 Mutable references and evaluation contexts [10 pts]

Consider the following definitions for simply-typed λ-calculus extended with mutable references:

<table>
<thead>
<tr>
<th>Type</th>
<th>Expression</th>
<th>Value</th>
<th>Store</th>
<th>Typing Context</th>
<th>Store Typing Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A ) ::=  ( P )</td>
<td>( e ) ::=  ( x )</td>
<td>( v ) ::= ( \lambda x : A. e )</td>
<td>( \psi ::= \cdot )</td>
<td>( \Gamma ::= \cdot )</td>
<td>( \Psi ::= \cdot )</td>
</tr>
<tr>
<td>( A \to A )</td>
<td>( e_1 e_2 )</td>
<td>( e ) ::= ( \lambda x : A. e )</td>
<td>( \psi, l \mapsto v )</td>
<td>( \Gamma, x : A )</td>
<td>( \Psi, l \mapsto A )</td>
</tr>
<tr>
<td>( \text{unit} )</td>
<td>()</td>
<td>()</td>
<td>()</td>
<td>()</td>
<td>()</td>
</tr>
<tr>
<td>( \text{ref } A )</td>
<td>( \text{ref } e )</td>
<td>( \text{ref } v )</td>
<td>( \text{ref } l \mapsto v )</td>
<td>( \text{ref } ! e )</td>
<td>( \text{ref } ! l )</td>
</tr>
</tbody>
</table>

In this problem, we use the following judgments:

- A typing judgment \( \Gamma \mid \Psi \vdash e : A \) means that expression \( e \) has type \( A \) under typing context \( \Gamma \) and store typing context \( \Psi \).

- A reduction judgment \( e \mid \psi \mapsto e' \mid \psi' \) means that expression \( e \) with store \( \psi \) reduces to \( e' \) with \( \psi' \). The reduction rules are defined as follows:

```
\[
\begin{align*}
\frac{e_1 \mid \psi \mapsto e'_1 \mid \psi'}{e_1 e_2 \mid \psi \mapsto e'_1 e'_2 \mid \psi'} \quad &\text{Lam} \\
\frac{e_2 \mid \psi \mapsto e'_2 \mid \psi'}{(\lambda x : A. e) e_2 \mid \psi \mapsto (\lambda x : A. e) e'_2 \mid \psi'} \quad &\text{Arg} \\
\frac{e \mid \psi \mapsto e' \mid \psi'}{(\lambda x : A. e) \mid \psi \mapsto [v/x]e \mid \psi} \quad &\text{App} \\
\frac{\text{ref } e \mid \psi \mapsto \text{ref } e' \mid \psi'}{(\text{ref } e) \mid \psi \mapsto (\text{ref } e') \mid \psi'} \quad &\text{Ref} \\
\frac{\text{ref } v \mid \psi \mapsto l \mid \psi, l \mapsto v \mid \psi'}{(\text{ref } v) \mid \psi \mapsto l \mid \psi, l \mapsto v \mid \psi'} \quad &\text{Ref'} \\
\frac{\text{! } e \mid \psi \mapsto \text{! } e' \mid \psi'}{(\text{! } e) \mid \psi \mapsto (\text{! } e') \mid \psi'} \quad &\text{Deref} \\
\frac{\psi(l) = v \mid \psi \mapsto v \mid \psi}{\psi(l) \mid \psi \mapsto v \mid \psi'} \quad &\text{Deref'} \\
\frac{e \mid \psi \mapsto e'' \mid \psi'}{e : = e' \mid \psi \mapsto e'' : = e' \mid \psi'} \quad &\text{Assign} \\
\frac{\text{! } e \mid \psi \mapsto \text{! } e' \mid \psi'}{\text{! } l \mid \psi \mapsto l \mid \psi \mapsto \text{! } e' \mid \psi'} \quad &\text{Assign'} \\
\frac{\text{! } e \mid \psi \mapsto (\cdot) \mid \psi}{\text{! } l \mid \psi \mapsto (\cdot) \mid \psi \mapsto l \mid \psi} \quad &\text{Assign''} \\
\end{align*}
\]
```

- A store judgment \( \psi \ ::= \Psi \) means that store typing context \( \Psi \) corresponds to store \( \psi \), or simply, \( \psi \) is well-typed with \( \Psi \). The formal definition is as follows:

\[
\begin{align*}
\text{dom}(\Psi) = \text{dom}(\psi) \quad &\cdot \quad \Psi \vdash \psi(l) : \Psi(l) \quad \text{for every } l \in \text{dom}(\psi) \\
\psi ::= \Psi \\
\end{align*}
\]

We write \( \text{dom}(\psi) \) for the domain of \( \psi \), i.e., the set of locations mapped to certain values under \( \psi \). Formally we define \( \text{dom}(\psi) \) as follows:

\[
\begin{align*}
\text{dom}(\cdot) &= \emptyset \\
\text{dom}(\psi, l \mapsto v) &= \text{dom}(\psi) \cup \{l\}
\end{align*}
\]
We write \([l \mapsto v]\psi\) for the store obtained by updating the contents of \(l\) in \(\psi\) with \(v\). Note that in order for \([l \mapsto v]\psi\) to be defined, \(l\) must be in \(\text{dom}(\psi)\):

\[
[l \mapsto v](\psi', l \mapsto v') = \psi', l \mapsto v
\]

We write \(\psi(l)\) for the value to which \(l\) is mapped under \(\psi\); in order for \(\psi(l)\) to be defined, \(l\) must be in \(\text{dom}(\psi)\):

\[
(\psi', l \mapsto v)(l) = v
\]

**Question 1. [4 pts]** State progress and type preservation theorems:

**Theorem 1.1 (Progress).**

Suppose that expression \(e\) satisfies \(\cdot | \Psi |- e : A\) for some store typing context \(\Psi\) and type \(A\). Then either:

1. \(e\) is a value, or
2. for any store \(\psi\) such that \(\psi :: \Psi\), there exist some expression \(e'\) and store \(\psi'\) such that \(e | \psi \mapsto e | \psi'\).

**Theorem 1.2 (Type preservation).**

Suppose \(\left\{\begin{array}{l}
\Gamma | \Psi |- e : A \\
\psi :: \Psi \\
e | \psi \mapsto e' | \psi'
\end{array}\right.\)

Then there exists a store typing context \(\Psi'\) such that

\[
\left\{\begin{array}{l}
\Gamma | \Psi' |- e' : A \\
\Psi \subset \Psi' \\
\psi' :: \Psi'
\end{array}\right.
\]
In class, we learned how to rewrite an expression as a pair of an evaluation context \( \kappa \) (an expression with a hole in it) and a redex. We also defined the call-by-value operational semantics using evaluation contexts for the simply-typed \( \lambda \)-calculus. We write \( \kappa[e] \) for the expression obtained by filling the hole in evaluation context \( \kappa \) with expression \( e \).

In this problem, we expand the idea of using evaluation contexts to deal with mutable references.

**Question 2. [2 pts]** Complete the definition of the evaluation context \( \kappa \) that corresponds to the operational semantics based on the call-by-value reduction strategy:

\[
\kappa ::= \square \mid \kappa e \mid (\lambda x : A. e) \kappa \mid \text{ref} \kappa \mid !\kappa \mid \kappa := e \mid l := \kappa
\]

**Question 3. [4 pts]** Define the operational semantics using \( \kappa[e] \) with as many reduction rules as you need. In your reduction rules, you may use the following relation \( \mapsto \beta \) for reducing redexes:

\[
(\lambda x : A. e) v \mapsto \beta [v/x]e
\]

\[
e \mapsto \beta e'
\]

\[
\kappa[e] \psi \mapsto \kappa[e'] \psi
\]

\[
l \notin \text{dom}(\psi)
\]

\[
\kappa[\text{ref} v] \psi \mapsto \kappa[l] \psi, l \mapsto v
\]

\[
\psi(l) = v
\]

\[
\kappa[l := v] \psi \mapsto \kappa(0)[l \mapsto v] \psi
\]
2 Environments and closures [11 pts]

In this problem, we design an abstract machine $E$ which allows a fixed point construct $\text{fun } f \; x : A.\; e$ and follows the call-by-name reduction strategy. We use the following definitions:

- **type** $A ::= P \mid A \rightarrow A$
- **expression** $e ::= x \mid \lambda x : A. \; e \mid e \; e \mid \text{fun } f \; x : A.\; e$
- **value** $v ::= \ldots$
- **environment** $\eta ::= \ldots$
- **frame** $\phi ::= \ldots$
- **stack** $\sigma ::= \emptyset \mid \sigma; \phi$
- **state** $s ::= \sigma \uparrow e @ \eta \mid \sigma \downarrow v$

In the definition of state $s$:

- $\sigma \uparrow e @ \eta$ means that the machine is currently analyzing $e$ under the environment $\eta$.
- $\sigma \downarrow v$ means that the machine is currently returning $v$ to the stack $\sigma$.

The transition judgment for the abstract machine $E$ is as follows:

\[
s \xrightarrow{E} s' \iff \text{the machine makes a transition from state } s \text{ to another state } s'
\]

Complete the definitions of value $v$, environment $\eta$, and frame $\phi$. Then define transition rules for the abstract machine $E$. You may introduce as many transition rules as you need. Explain your definitions and reduction rules.

(Definitions)

- **value** $v ::= [\eta, \lambda x : A.\; e] \mid [\eta, \text{fun } f \; x : A.\; e]$
- **environment** $\eta ::= \cdot \mid \eta, x \mapsto \text{delayed}(e, \eta) \mid \eta, f \mapsto [\eta', \text{fun } f \; x : A.\; e]$
- **frame** $\phi ::= \Box_\eta e$
(Transition rules)

\[
\begin{align*}
\frac{x \mapsto \text{delayed}(e, \eta') \in \eta}{\sigma \triangleright x \odot \eta \mapsto_{E} \sigma \triangleright e \odot \eta'} &\quad \Var_{E} \\
\frac{\sigma \triangleright \lambda x : A. e \odot \eta \mapsto_{E} \sigma \triangleright [\eta, \lambda x : A. e]}{\sigma} &\quad \text{Closure}_{E} \\
\frac{\sigma \triangleright e_{1} \cdot e_{2} \odot \eta \mapsto_{E} \sigma ; \Box_{\eta} e_{2} \triangleright e_{1} \odot \eta}{\sigma \triangleright} &\quad \text{Lam}_{E} \\
\frac{\sigma ; \Box_{\eta} e_{2} \triangleleft [\eta', \lambda x : A. e] \mapsto_{E} \sigma \triangleright e \odot \eta', x \mapsto \text{delayed}(e_{2}, \eta)}{\sigma \triangleright e_{1} \odot \eta \mapsto_{E} \sigma ; e \odot \eta', x \mapsto \text{delayed}(e_{2}, \eta)} &\quad \text{App}_{E} \\
\frac{\sigma ; \Box_{\eta} e_{2} \triangleleft [\eta', \text{fun } f \cdot x : A. e] \mapsto_{E} \sigma \triangleright e \odot \eta', f \mapsto [\eta', \text{fun } f \cdot x : A. c], x \mapsto \text{delayed}(e_{2}, \eta)}{\sigma \triangleright} &\quad \text{App}_{E}^{R} \\
\frac{\sigma \triangleright \text{fun } f \cdot x : A. e \odot \eta \mapsto_{E} \sigma \triangleright [\eta, \text{fun } f \cdot x : A. c]}{\sigma \triangleright} &\quad \text{Closure}_{E}^{R}
\end{align*}
\]
3 Abstract machine N [11 pts]

4 Subtyping [10 pts]

Consider the following definitions for the simply-typed \( \lambda \)-calculus:

\[
\begin{align*}
\text{type} & :\quad A ::= P \mid A \rightarrow A \mid A \times A \\
\text{expression} & :\quad e ::= x \mid \lambda x:A.e \mid e \; e \mid (e,e) \mid \text{fst} \; e \mid \text{snd} \; e \\
\text{typing context} & :\quad \Gamma ::= \cdot \mid \Gamma, x : A
\end{align*}
\]

We write \( A \leq B \) if \( A \) is a subtype of \( B \), or equivalently, if \( B \) is a supertype of \( A \). We also use a typing judgment \( \Gamma \vdash x : A \).

**Question 1. [2 pts]** Write the subtyping rule for function types:

\[
\frac{A' \leq A \quad B \leq B'}{A \rightarrow B \leq A' \rightarrow B'} \quad \text{Fun}_\leq
\]

**Question 2. [2 pts]** Write the rule of subsumption:

The rule of subsumption is a typing rule which enables us to change the type of an expression to its supertype:

\[
\frac{\Gamma \vdash e : A \quad A \leq B}{\Gamma \vdash e : B} \quad \text{Sub}
\]
Question 3. [6 pts] 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 $A \rightarrow B$. 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 $A \rightarrow B$. For example, a judgment $\text{int} \leq \text{float} \Rightarrow \text{int2float}$ holds if the coercion function $\text{int2float}$ converts integers of type int to floating point numbers of type float.

The following is a subtyping system for the coercion semantics. The rules $\text{Refl}_C^\leq$ and $\text{Trans}_C^\leq$ express reflexivity and transitivity of the subtyping relation, respectively. Define the subtyping rules for product types and function types:

$$\frac{A \leq A \Rightarrow \lambda x:A. x}{\text{Refl}_C^\leq} \quad \frac{A \leq B \Rightarrow f \quad B \leq C \Rightarrow g}{A \leq C \Rightarrow \lambda x:A. g \,(f \,x)} \quad \text{Trans}_C^\leq$$

$$\frac{A \leq A' \Rightarrow f \quad B \leq B' \Rightarrow g}{A \times B \leq A' \times B' \Rightarrow \lambda x:A \times B. \,(f \,(\text{fst} \,x), g \,(\text{snd} \,x))} \quad \text{Prod}_C^\leq$$

$$\frac{A' \leq A \Rightarrow f \quad B \leq B' \Rightarrow g}{A \rightarrow B \leq A' \rightarrow B' \Rightarrow \lambda h:A \rightarrow B. \lambda x:A'. g \,(h \,(f \,x))} \quad \text{Fun}_C^\leq$$
5 Recursive types [7 pts]

Consider the simply-typed λ-calculus with product types, sum types, unit type, base type \text{nat}, recursive types, and the fixed point construct:

| type | $A ::= A \to A \mid A \times A \mid A + A \mid \alpha \mid \mu\alpha.A \mid \text{unit} \mid \text{nat}$ |
| expression | $e ::= x \mid \lambda x : A. e \mid e e \mid \text{fix } x : A. e \mid (e, e) \mid \text{fst } e \mid \text{snd } e \mid \text{inl } A e \mid \text{inr } A e \mid \text{fold}_C e \mid \text{unfold}_C e \mid () \mid + \mid - \mid 0 \mid 1 \mid \cdots$ |
| typing context | $\Gamma ::= \cdot \mid \Gamma, x : A \mid \Gamma, \alpha$ |
| value | $v ::= \lambda x : A. e \mid (v, v) \mid \text{inl } A v \mid \text{inr } A v \mid () \mid \text{fold}_C v \mid + \mid - \mid 0 \mid 1 \mid \cdots$ |

+ and − are functions for arithmetic addition and subtraction, respectively. 0, 1, · · · are integer constants.

**Question 1.** [4 pts] Translate the following definition in SML for lists of natural numbers into the simply-typed λ-calculus with recursive types.

```sml
datatype nlist = Nil | Cons of nat × nlist

nlist = μα. unit+ (nat × α)

Nil = foldnlist inlhonnlist ()

Cons e = foldnlist inrunit e

case e of Nil ⇒ e₁ | Cons x ⇒ e₂ = case unfoldnlist e of inl x.e₁ | inr x.e₂
```

**Question 2.** [3 pts] In this question, we define a datatype for streams of natural numbers, that is, \text{nstream}. A formal definition of \text{nstream} is as follows:

\text{nstream} = µα. \text{unit}→\text{nat}×α

When “unfolded,” a value of type \text{nstream} yields a function of type unit→\text{nat}×\text{nstream} which returns a natural number and another stream. For example, the following λ-abstraction has type \text{nstream}→\text{nat}×\text{nstream}:

\[\lambda s: \text{nstream}. \text{unfold}_\text{nstream} s ()\]

Define a function \(f\) of type \text{nat}→\text{nstream} that returns a stream of natural numbers beginning with its argument. For example, \(f\ n\) returns the stream \(\{n, n + 1, n + 2, \cdots\}\).

\[f = \lambda n: \text{nat}. (\text{fix } f: \text{nat}→\text{nstream}. \lambda x: \text{nat}. \text{fold}_\text{nstream} \lambda y: \text{unit}. (x, f (+ (x, 1)))) n\]
6 System F [17 pts]

Consider the following definitions for System F:

**type**

\[ A ::= A \to A \mid \alpha \mid \forall \alpha.A \]

**expression**

\[ e ::= x \mid \lambda x : A . e \mid e e \mid \Lambda \alpha . e \mid e [A] \]

**value**

\[ v ::= \lambda x : A . e \mid \Lambda \alpha . e \]

**typing context**

\[ \Gamma ::= \cdot \mid \Gamma , x : A \mid \Gamma , \alpha \text{ type} \]

Note that a typing context \( \Gamma \) is an *ordered* set of type bindings and type declarations.

We use three judgments: a reduction judgment, a type judgment, and a typing judgment.

\[ e \mapsto e' \iff e \text{ reduces to } e' \]

\[ \Gamma \vdash A \text{ type} \iff A \text{ is a valid type with respect to typing context } \Gamma \]

\[ \Gamma \vdash e : A \iff e \text{ has type } A \text{ under typing context } \Gamma \]

**Question 1. [2 pts]** Write the reduction rules for type applications:

\[
\frac{e \mapsto e'}{e[A] \mapsto e'[A]} Tlam \quad (\Lambda \alpha . e)[A] \mapsto [A/\alpha]e Tapp
\]

**Question 2. [2 pts]** Write the typing rules for type abstractions and type applications:

\[
\frac{\Gamma, \alpha \text{ type } \vdash e : A}{\Gamma \vdash \Lambda \alpha . e : \forall \alpha . A} \forall I \quad \frac{\Gamma \vdash e : \forall \alpha . B \quad \Gamma \vdash A \text{ type}}{\Gamma \vdash e[A] : [A/\alpha]B} \forall E
\]
**Question 3. [6 pts]** In order to prove type preservation of the simply-typed λ-calculus, we introduced the substitution lemma. The proof of type safety of System F needs three substitution lemmas because there are three kinds of substitutions in System F: type substitution into types, type substitution into expressions, and expression substitution.

State the substitution lemmas for the proof of type safety of System F:

(for substituting types for type variables in types)

If \( \Gamma \vdash A \text{ type} \) and \( \Gamma, \alpha \text{ type} \), \( \Gamma' \vdash B \text{ type} \), then \( \Gamma, [A/\alpha] \Gamma' \vdash [A/\alpha] B \text{ type} \).

(for substituting types for type variables in expressions)

If \( \Gamma \vdash A \text{ type} \) and \( \Gamma, \alpha \text{ type} \), \( \Gamma' \vdash e : B \), then \( \Gamma, [A/\alpha] \Gamma' \vdash [A/\alpha] e : [A/\alpha] B \).

(for substituting expressions for variables in expressions)

If \( \Gamma \vdash e : A \) and \( \Gamma, x : A, \Gamma' \vdash e' : C \), then \( \Gamma, \Gamma' \vdash [e/x] e' : C \).

**Question 4. [3 pts]** Encode a product type \( A \times B \), \text{pair}, and \text{fst} in System F:

\[
A \times B = \forall \alpha.(A \rightarrow B \rightarrow \alpha) \rightarrow \alpha
\]

\[
\text{pair} : \forall \alpha \forall \beta. \alpha \rightarrow \beta \rightarrow \alpha \times \beta = \Lambda \alpha. \Lambda \beta. \lambda x : \alpha. \lambda y : \beta. \Lambda \gamma. \lambda f : \alpha \rightarrow \beta \rightarrow \gamma. f \; x \; y
\]

\[
\text{fst} : \forall \alpha \forall \beta. \alpha \times \beta \rightarrow \alpha = \Lambda \alpha. \Lambda \beta. \lambda p : \alpha \times \beta. p \llbracket \alpha \rrbracket (\lambda x : \alpha. \lambda y : \beta. x)
\]

**Question 5. [4 pts]** Explain why System F is called an *impredicative* polymorphic λ-calculus, not a predicative polymorphic λ-calculus:

*impredicative* polymorphism - allows type variables to range over polymorphic types; type variables can be substituted all kinds of types including polymorphic types.

*predicative* polymorphism - prohibits type variables from being substituted by polymorphic types; type substitutions accept only monomorphic types.

System F is impredicative polymorphic because a type \( A \) in a type application \( e \llbracket A \rrbracket \) ranges over polymorphic types.
7 Predicative polymorphic λ-calculus [4 pts]

Consider the following definitions for the predicative polymorphic λ-calculus:

monotype \( A ::= A \rightarrow A | \alpha \)

polytype \( U ::= A | \forall \alpha.U \)

equation \( e ::= x | \lambda x: A. e | e e | \Lambda \alpha. e \mid e[A] \)

equation \( v ::= \lambda x: A. e | \Lambda \alpha. e \)

typing context \( \Gamma ::= \cdot | \Gamma, x: A | \Gamma, \alpha \text{ type} \)

Question 1. [2 pts] Write the typing rules for type abstractions and type applications:

\[ \frac{\Gamma, \alpha \text{ type } \vdash e : U \forall I}{\Gamma \vdash \Lambda \alpha. e : \forall \alpha.U} \quad \frac{\Gamma \vdash e : \forall \alpha.U \quad \Gamma \vdash A \text{ type}}{\Gamma \vdash e[A] : [A/\alpha]U} \] \( \forall E \)

Question 2. [2 pts] Give an expression in the untyped λ-calculus that is typable in System F but not in the predicative polymorphic λ-calculus. You may use the following constructs in your solution:

- \((e_1, e_2)\) builds a pair of expressions \(e_1\) and \(e_2\).
- \text{true} has type \text{bool} in both System F and the predicative polymorphic λ-calculus.
- \text{0} has type \text{int} in both System F and the predicative polymorphic λ-calculus.

\[(\lambda f. ((\text{true}, f \text{ 0}))) (\lambda x. x)\]
8 Type reconstruction [8 pts]

In this problem, we study the design of a type reconstruction algorithm. We assume an untyped language $L_u$, a typed language $L_t$, and a type reconstruction algorithm $Y$.

1) untyped language $L_u$

- syntax:
  
  untyped expression $e ::= \cdots$

- reduction judgment:
  
  $e \rightarrow e' \iff e \text{ reduces to } e'$

2) typed language $L_t$

- syntax:
  
  type $A ::= \cdots$
  typed expression $t ::= \cdots$

- typing judgment:
  
  $t : A \iff t \text{ has type } A$

- reduction judgment:
  
  $t \Rightarrow t' \iff t \text{ reduces to } t'$

3) type reconstruction algorithm $Y$

- input: an untyped expression $e$ in $L_u$

- output: a typed expression $t$ and a type $A$ in $L_t$ if the input $e$ is typable, and failure otherwise.

Suppose that the algorithm $Y$ produces a typed expression $t$ and a type $A$ from an untyped expression $e$. Explain what conditions on $e$, $t$, and $A$ are necessary in order for $Y$ to be eligible for a type reconstruction algorithm.

For this, we assume a function $erase(t)$ that takes a typed expression $t$ and removes all type annotations in it. $e$, $t$, and $A$ should satisfy the following conditions:

- $t : A$
- $erase(t) = e$
- if $t \Rightarrow t'$, then there exists an untyped expression $e'$ such that $erase(t') = e'$ and $e \rightarrow^* e'$
9 Value restriction [6 pts]

The interaction between polymorphism and computational effects such as mutable references makes a naive type reconstruction algorithm unsound. SML solves this problem with value restriction on let-bindings.

Consider the following three SML expressions:

(** expression 1 **)  
let val id = (fn y => y) (fn z => z) in id true end

(** expression 2 **)  
let val id = (fn y => y) (fn z => z) in (id true, id 0) end

(** expression 3 **)  
let val id = (fn y => y) (fn z => z) in id end

Each expression either typechecks, raises a type error, or prints a warning message. State and explain the result of typechecking each expression.

(expression 1) : typechecks because id is monomorphically used.

(expression 2) : does not typecheck. The value restriction prohibits any non-value expression from having a polytype, and id binds to (fn y => y) (fn z => z) which is not a value, but in (id true, id 0), id is polymorphically used.

(expression 3) : typechecks, but prints a warning message because the typechecking algorithm cannot infer the type of id.
10 The algorithm \( \mathcal{W} \) [16 pts]

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

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>monotype ( A )</td>
<td>( A := A \rightarrow A \mid \alpha )</td>
</tr>
<tr>
<td>polytype ( U )</td>
<td>( U ::= A \mid \forall \alpha. U )</td>
</tr>
<tr>
<td>expression ( e )</td>
<td>( e ::= x \mid \lambda x. e \mid e \ e \mid \text{let } x = e \text{ in } e )</td>
</tr>
<tr>
<td>typing context ( \Gamma )</td>
<td>( \Gamma ::= \cdot \mid \Gamma, x : U )</td>
</tr>
<tr>
<td>type substitution ( S )</td>
<td>( S ::= \text{id} \mid { A/\alpha } \mid S \circ S )</td>
</tr>
<tr>
<td>type equations ( E )</td>
<td>( E ::= \cdot \mid E, A = A )</td>
</tr>
</tbody>
</table>

We use the following auxiliary functions and notations:

- \( S \cdot U \) and \( S \cdot \Gamma \) denote applications of \( S \) to \( U \) and \( \Gamma \), respectively.
- \( \text{ftv}(\Gamma) \) denotes the set of free type variables in \( \Gamma \); \( \text{ftv}(U) \) denotes the set of free type variables in \( U \).
- We write \( \Gamma + x : U \) for \( \Gamma - \{ x : U' \} \), \( x : U \) if \( x : U' \in \Gamma \), and for \( \Gamma, x : U \) if \( \Gamma \) contains no type binding for variable \( x \).

We use a typing judgment \( \Gamma \vdash e : U \) to express that untyped expression \( e \) is typable with polytype \( U \). The typing rules for the typing judgment \( \Gamma \vdash e : U \) are as follows:

\[
\begin{array}{c}
\frac{x : U \in \Gamma}{\Gamma \vdash x : U} \quad \text{Var} \\
\frac{\Gamma \vdash x : A \triangleleft e : B}{\Gamma \vdash \lambda x. e : A \rightarrow B} \quad \rightarrow I \\
\frac{\Gamma \vdash e : A \rightarrow B}{\Gamma \vdash e' : A} \quad \rightarrow E \\
\frac{\Gamma \vdash e : U \quad \Gamma, x : U \vdash e' : A}{\Gamma \vdash \text{let } x = e \text{ in } e' : A} \quad \text{Let} \\
\frac{\Gamma \vdash e : U \quad \alpha \notin \text{ftv}(\Gamma)}{\Gamma \vdash e : \forall \alpha.U} \quad \text{Gen} \\
\frac{\Gamma \vdash e : \forall \alpha.U}{\Gamma \vdash e : [A/\alpha]U} \quad \text{Spec}
\end{array}
\]
Question 1. [3 pts] Unify($E$) is a function that attempts to calculate a type substitution that unifies two types $A$ and $A'$ in every type equation $A = A'$ in $E$. If no such type substitution exists, Unify($E$) returns fail. Complete the definition of Unify($E$).

\[
\text{Unify}(\cdot) = \text{id}
\]

\[
\text{Unify}(E, \alpha = A) = \text{Unify}(E, A = \alpha) = \begin{cases} 
\text{if } \alpha = A \text{ then } \text{Unify}(E) \\
\text{else if } \alpha \in \text{ftv}(A) \text{ then } \text{fail} \\
\text{else } \text{Unify}({\{A/\alpha\}} \cdot E) \circ {\{A/\alpha\}} 
\end{cases}
\]

\[
\text{Unify}(E, A_1 \rightarrow A_2 = B_1 \rightarrow B_2) = \text{Unify}(E, A_1 = B_1, A_2 = B_2)
\]

Question 2. [4 pts] Write the result of applying the function Gen$_\Gamma(A)$ which generalizes monotype $A$ to a polytype after taking into account free type variables in typing context $\Gamma$:

\[
\text{Gen}_\cdot(\alpha \rightarrow \alpha) = \forall \alpha. \alpha \rightarrow \alpha
\]

\[
\text{Gen}_{x.\alpha}(\alpha \rightarrow \alpha) = \alpha \rightarrow \alpha
\]

\[
\text{Gen}_{x.\alpha}(\alpha \rightarrow \beta) = \forall \beta. \alpha \rightarrow \beta
\]

\[
\text{Gen}_{x.\alpha,y.\beta}(\alpha \rightarrow \beta) = \alpha \rightarrow \beta
\]
**Question 3. [7 pts]** The type reconstruction algorithm $W$ takes a typing context $\Gamma$ and an expression $e$ as input, and returns a pair of a type substitution $S$ and a monotype $A$ as output:

$$W(\Gamma, e) = (S, A)$$

Complete the definition of the algorithm $W$:

$$W(\Gamma, x) = (\text{id}, \{\vec{\beta}/\vec{\alpha}\} \cdot A)$$

$$W(\Gamma, \lambda x. e) = \text{let } (S, A) = W(\Gamma + x : \alpha, e) \text{ in } (S, (S \cdot \alpha) \rightarrow A)$$

$$W(\Gamma, e_1 e_2) = \text{let } (S_1, A_1) = W(\Gamma, e_1) \text{ in }$$

$$\text{let } (S_2, A_2) = W(S_1 \cdot \Gamma, e_2) \text{ in }$$

$$\text{let } S_3 = \text{Unify}(S_2 \cdot A_1 = A_2 \rightarrow \alpha) \text{ in } \text{fresh } \alpha$$

$$\text{(S}_3 \circ S_2 \circ S_1, S_3 \cdot \alpha)$$

$$W(\Gamma, \text{let } x = e_1 \text{ in } e_2) = \text{let } (S_1, A_1) = W(\Gamma, e_1) \text{ in }$$

$$\text{let } (S_2, A_2) = W(S_1 \cdot \Gamma + x : \text{Gen}_{S_1, \Gamma}(A_1), e_2) \text{ in }$$

$$\text{(S}_2 \circ S_1, A_2)$$

**Question 4. [2 pts]** State the soundness theorem of the algorithm $W$:

(Soundness of $W$)

If $W(\Gamma, e) = (S, A)$, then $S \cdot \Gamma \vdash e : A$. 

Work sheet