Name: Hemos ID: Score: / 100

CSE-321 Programming Languages 2012
Final — Sample Solution

Prob 1	Prob 2	Prob 3	Prob 4	Prob 5	Prob 6	Prob 7	Prob 8	Prob 9	Prob 10
10	11	11	10	7	17	4	8	6	16

- 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:

$$
\begin{array}{rrcl}
\textsf{type} & A & ::= & P \mid A \to A \mid \textsf{unit} \mid \textsf{ref}\ A \\
\textsf{expression} & e & ::= & x \mid \lambda x\!:\!A.\, e \mid e\ e \mid () \mid \textsf{ref}\ e \mid !e \mid e := e \mid l \\
\textsf{value} & v & ::= & \lambda x\!:\!A.\, e \mid () \mid l \\
\textsf{store} & \psi & ::= & \cdot \mid \psi, l \mapsto v \\
\textsf{typing context} & \Gamma & ::= & \cdot \mid \Gamma, x : A \\
\textsf{store typing context} & \Psi & ::= & \cdot \mid \Psi, l \mapsto A
\end{array}
$$

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 Γ and store typing context Ψ.
- A reduction judgment $e \mid \psi \mapsto e' \mid \psi'$ means that expression e with store ψ reduces to e' with ψ'. The reduction rules are defined as follows:

$$
\frac{e_1 \mid \psi \mapsto e_1' \mid \psi'}{e_1\ e_2 \mid \psi \mapsto e_1'\ e_2 \mid \psi'}\ \textit{Lam}
$$

$$
\frac{e_2 \mid \psi \mapsto e_2' \mid \psi'}{(\lambda x\!:\!A.\, e)\ e_2 \mid \psi \mapsto (\lambda x\!:\!A.\, e)\ e_2' \mid \psi'}\ \textit{Arg}
\qquad
\frac{}{(\lambda x\!:\!A.\, e)\ v \mid \psi \mapsto [v/x]e \mid \psi}\ \textit{App}
$$

$$
\frac{e \mid \psi \mapsto e' \mid \psi'}{\textsf{ref}\ e \mid \psi \mapsto \textsf{ref}\ e' \mid \psi'}\ \textit{Ref}
\qquad
\frac{l \notin \textit{dom}(\psi)}{\textsf{ref}\ v \mid \psi \mapsto l \mid \psi, l \mapsto v}\ \textit{Ref}'
$$

$$
\frac{e \mid \psi \mapsto e' \mid \psi'}{!e \mid \psi \mapsto !e' \mid \psi'}\ \textit{Deref}
\qquad
\frac{\psi(l) = v}{!l \mid \psi \mapsto v \mid \psi}\ \textit{Deref}'
$$

$$
\frac{e \mid \psi \mapsto e'' \mid \psi'}{e := e' \mid \psi \mapsto e'' := e' \mid \psi'}\ \textit{Assign}
$$

$$
\frac{e \mid \psi \mapsto e' \mid \psi'}{l := e \mid \psi \mapsto l := e' \mid \psi'}\ \textit{Assign}'
\qquad
\frac{}{l := v \mid \psi \mapsto () \mid [l \mapsto v]\psi}\ \textit{Assign}''
$$

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

$$
\frac{\textit{dom}(\Psi) = \textit{dom}(\psi) \quad \cdot \mid \Psi \vdash \psi(l) : \Psi(l) \ \text{ for every } l \in \textit{dom}(\psi)}{\psi :: \Psi}\ \textsf{Store}
$$

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

$$
\begin{array}{rcl}
\textit{dom}(\cdot) & = & \varnothing \\
\textit{dom}(\psi, l \mapsto v) & = & \textit{dom}(\psi) \cup \{l\}
\end{array}
$$

We write $[l \mapsto v]\psi$ for the store obtained by updating the contents of l in ψ with v. Note that in order for $[l \mapsto v]\psi$ to be defined, l must be in $dom(\psi)$:

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

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

$$(\psi', l \mapsto v)(l) \quad = \quad v$$

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

Theorem 1.1 (Progress).
Suppose that expression e satisfies $\cdot \mid \Psi \vdash e : A$ for some store typing context Ψ and type A. Then either:

(1) e is a value, or

(2) for any store ψ such that $\psi :: \Psi$,

there exist some expression e' and store ψ' such that $e \mid \psi \mapsto e' \mid \psi'$.

Theorem 1.2 (Type preservation).

Suppose $\begin{cases} \Gamma \mid \Psi \vdash e : A \\ \psi :: \Psi \\ e \mid \psi \mapsto e' \mid \psi' \end{cases}$.

Then there exists a store typing context Ψ' such that $\begin{cases} \Gamma \mid \Psi' \vdash e' : A \\ \Psi \subset \Psi' \\ \psi' :: \Psi' \end{cases}$.

In class, we learned how to rewrite an expression as a pair of an evaluation context κ (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 λ-calculus. We write $\kappa[\![e]\!]$ for the expression obtained by filling the hole in evaluation context κ 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 κ that corresponds to the operational semantics based on the call-by-value reduction strategy:

$$\text{evaluation context} \qquad \kappa \quad ::= \quad \underline{\square \mid \kappa\ e \mid (\lambda x\!:\!A.\,e)\ \kappa \mid \mathsf{ref}\ \kappa \mid !\kappa \mid \kappa := e \mid l := \kappa}$$

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:

$$(\lambda x\!:\!A.\,e)\ v \quad \mapsto_\beta \quad [v/x]e$$

$$\frac{e \mapsto_\beta e'}{\kappa[\![e]\!]|\psi \mapsto \kappa[\![e']\!]|\psi}$$

$$\frac{l \notin dom(\psi)}{\kappa[\![\mathsf{ref}\ v]\!]|\psi \mapsto \kappa[\![l]\!]|\psi, l \mapsto v}$$

$$\frac{\psi(l) = v}{\kappa[\![!l]\!]|\psi \mapsto \kappa[\![v]\!]|\psi}$$

$$\frac{}{\kappa[\![l := v]\!]|\psi \mapsto \kappa[\![()]\!]|[l \mapsto v]\psi}$$

2 Environments and closures [11 pts]

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

$$
\begin{array}{rrcl}
\textsf{type} & A & ::= & P \mid A \to A \\
\textsf{expression} & e & ::= & x \mid \lambda x : A.\, e \mid e\ e \mid \mathsf{fun}\ f\ x : A.\, e \\
\textsf{value} & v & ::= & \underline{\hspace{6cm}} \\
\textsf{environment} & \eta & ::= & \underline{\hspace{6cm}} \\
\textsf{frame} & \phi & ::= & \underline{\hspace{6cm}} \\
\textsf{stack} & \sigma & ::= & \square \mid \sigma ; \phi \\
\textsf{state} & s & ::= & \sigma \blacktriangleright e \mathbin{@} \eta \mid \sigma \blacktriangleleft v
\end{array}
$$

In the definition of state s:

- $\sigma \blacktriangleright e \mathbin{@} \eta$ means that the machine is currently analyzing e under the environment η.
- $\sigma \blacktriangleleft v$ means that the machine is currently returning v to the stack σ.

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

$$s \mapsto_{\mathsf{E}} s' \quad \Leftrightarrow \quad \textit{the machine makes a transition from state } s \textit{ to another state } s'$$

Complete the definitions of value v, environment η, and frame ϕ. 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)

$$\textsf{value} \quad v \ ::= \ \underline{[\eta, \lambda x : A.\, e] \mid [\eta, \mathsf{fun}\ f\ x : A.\, e]}$$

$$\textsf{environment} \quad \eta \ ::= \ \underline{\cdot \mid \eta, x \hookrightarrow \mathsf{delayed}(e, \eta) \mid \eta, f \hookrightarrow [\eta', \mathsf{fun}\ f\ x : A.\, e]}$$

$$\textsf{frame} \quad \phi \ ::= \ \underline{\square_{\eta}\ e}$$

(Transition rules)

$$\frac{x \hookrightarrow \mathsf{delayed}(e, \eta') \in \eta}{\sigma \blacktriangleright x \mathbin{@} \eta \mapsto_{\mathsf{E}} \sigma \blacktriangleright e \mathbin{@} \eta'}\ \mathit{Var}_{\mathsf{E}}$$

$$\frac{}{\sigma \blacktriangleright \lambda x\!:\!A.\, e \mathbin{@} \eta \mapsto_{\mathsf{E}} \sigma \blacktriangleleft [\eta, \lambda x\!:\!A.\, e]}\ \mathit{Closure}_{\mathsf{E}}$$

$$\frac{}{\sigma \blacktriangleright e_1\ e_2 \mathbin{@} \eta \mapsto_{\mathsf{E}} \sigma; \square_\eta\ e_2 \blacktriangleright e_1 \mathbin{@} \eta}\ \mathit{Lam}_{\mathsf{E}}$$

$$\frac{}{\sigma; \square_\eta\ e_2 \blacktriangleleft [\eta', \lambda x\!:\!A.\, e] \mapsto_{\mathsf{E}} \sigma \blacktriangleright e \mathbin{@} \eta', x \hookrightarrow \mathsf{delayed}(e_2, \eta)}\ \mathit{App}_{\mathsf{E}}$$

$$\frac{}{\sigma; \square_\eta\ e_2 \blacktriangleleft [\eta', \mathsf{fun}\ f\ x\!:\!A.\, e] \mapsto_{\mathsf{E}} \sigma \blacktriangleright e \mathbin{@} \eta', f \hookrightarrow [\eta', \mathsf{fun}\ f\ x\!:\!A.\, e], x \hookrightarrow \mathsf{delayed}(e_2, \eta)}\ \mathit{App}^R_{\mathsf{E}}$$

$$\frac{}{\sigma \blacktriangleright \mathsf{fun}\ f\ x\!:\!A.\, e \mathbin{@} \eta \mapsto_{\mathsf{E}} \sigma \blacktriangleleft [\eta, \mathsf{fun}\ f\ x\!:\!A.\, e]}\ \mathit{Closure}^R_{\mathsf{E}}$$

3 Abstract machine N [11 pts]

4 Subtyping [10 pts]

Consider the following definitions for the simply-typed λ-calculus:

$$\begin{array}{rrcl} \textsf{type} & A & ::= & P \mid A \to A \mid A \times A \\ \textsf{expression} & e & ::= & x \mid \lambda x\!:\!A.\, e \mid e\ e \mid (e, e) \mid \textsf{fst}\ e \mid \textsf{snd}\ e \\ \textsf{typing context} & \Gamma & ::= & \cdot \mid \Gamma, x : A \end{array}$$

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:

$$\frac{A' \leq A \quad B \leq B'}{A \to B \leq A' \to B'}\ \mathit{Fun}_{\leq}$$

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:

$$\frac{\Gamma \vdash e : A \quad A \leq B}{\Gamma \vdash e : B}\ \textsf{Sub}$$

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 λ-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 $\mathsf{int} \leq \mathsf{float} \Rightarrow \mathsf{int2float}$ holds if the coercion function $\mathsf{int2float}$ converts integers of type int to floating point numbers of type float.

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

$$\frac{}{A \leq A \Rightarrow \lambda x\!:\!A.\, x}\ \mathit{Refl}_{\leq}^{\mathsf{C}} \qquad \frac{A \leq B \Rightarrow f \quad B \leq C \Rightarrow g}{A \leq C \Rightarrow \lambda x\!:\!A.\, g\ (f\ x)}\ \mathit{Trans}_{\leq}^{\mathsf{C}}$$

$$\frac{A \leq A' \Rightarrow f \quad B \leq B' \Rightarrow g}{A \times B \leq A' \times B' \Rightarrow \lambda x\!:\!A \times B.\, (f\ (\mathsf{fst}\ x), g\ (\mathsf{snd}\ x))}\ \mathit{Prod}_{\leq}^{\mathsf{C}}$$

$$\frac{A' \leq A \Rightarrow f \quad B \leq B' \Rightarrow g}{A \to B \leq A' \to B' \Rightarrow \lambda h\!:\!A \to B.\, \lambda x\!:\!A'.\, g\ (h\ (f\ x))}\ \mathit{Fun}_{\leq}^{\mathsf{C}}$$

5 Recursive types [7 pts]

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

$$
\begin{array}{rrcl}
\text{type} & A & ::= & A \rightarrow A \mid A \times A \mid A + A \mid \alpha \mid \mu\alpha.A \mid \mathsf{unit} \mid \mathsf{nat} \\
\text{expression} & e & ::= & x \mid \lambda x\!:\!A.\,e \mid e\ e \mid \mathsf{fix}\ x\!:\!A.\,e \mid \\
 & & & (e, e) \mid \mathsf{fst}\ e \mid \mathsf{snd}\ e \mid \\
 & & & \mathsf{inl}_A\ e \mid \mathsf{inr}_A\ e \mid \mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}\ x.\,e \mid \mathsf{inr}\ y.\,e \mid \\
 & & & \mathsf{fold}_C\ e \mid \mathsf{unfold}_C\ e \mid () \mid \\
 & & & + \mid - \mid 0 \mid 1 \mid \cdots \\
\text{typing context} & \Gamma & ::= & \cdot \mid \Gamma, x : A \mid \Gamma, \alpha\ \mathsf{type} \\
\text{value} & v & ::= & \lambda x\!:\!A.\,e \mid (v, v) \mid \mathsf{inl}_A\ v \mid \mathsf{inr}_A\ v \mid () \mid \mathsf{fold}_C\ v \mid + \mid - \mid 0 \mid 1 \mid \cdots
\end{array}
$$

$+$ and $-$ are functions for arithmetic addition and subtraction, respectively. 0, 1, $\cdots$ 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.

$$
\begin{array}{rcl}
\mathsf{datatype}\ \mathtt{nlist} & = & \mathtt{Nil} \mid \mathsf{Cons\ of}\ \mathtt{nat} \times \mathtt{nlist} \\
\mathtt{nlist} & = & \mu\alpha.\mathsf{unit} + (\mathtt{nat} \times \alpha) \\
\mathtt{Nil} & = & \mathsf{fold}_{\mathtt{nlist}}\ \mathsf{inl}_{\mathtt{nat} \times \mathtt{nlist}}\ () \\
\mathsf{Cons}\ e & = & \mathsf{fold}_{\mathtt{nlist}}\ \mathsf{inr}_{\mathsf{unit}}\ e \\
\mathsf{case}\ e\ \mathsf{of}\ \mathtt{Nil} \Rightarrow e_1 \mid \mathsf{Cons}\ x \Rightarrow e_2 & = & \mathsf{case}\ \mathsf{unfold}_{\mathtt{nlist}}\ e\ \mathsf{of}\ \mathsf{inl}\ \_.\,e_1 \mid \mathsf{inr}\ x.\,e_2
\end{array}
$$

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

$$\mathtt{nstream} = \mu\alpha.\mathsf{unit} \rightarrow \mathtt{nat} \times \alpha$$

When "unfolded," a value of type $\mathtt{nstream}$ yields a function of type $\mathsf{unit} \rightarrow \mathtt{nat} \times \mathtt{nstream}$ which returns a natural number and another stream. For example, the following λ-abstraction has type $\mathtt{nstream} \rightarrow \mathtt{nat} \times \mathtt{nstream}$:

$$\lambda s\!:\!\mathtt{nstream}.\,\mathsf{unfold}_{\mathtt{nstream}}\ s\ ()$$

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

$$f = \lambda n\!:\!\mathtt{nat}.\,(\mathsf{fix}\ f\!:\!\mathtt{nat} \rightarrow \mathtt{nstream}.\,\lambda x\!:\!\mathtt{nat}.\,\mathsf{fold}_{\mathtt{nstream}}\ \lambda y\!:\!\mathsf{unit}.\,(x, f\ (+\ (x, 1))))\ n$$

6 System F [17 pts]

Consider the following definitions for System F:

$$\begin{array}{rrcl} \textsf{type} & A & ::= & A \rightarrow A \mid \alpha \mid \forall \alpha. A \\ \textsf{expression} & e & ::= & x \mid \lambda x : A.\, e \mid e\ e \mid \Lambda \alpha.\, e \mid e\, [\![A]\!] \\ \textsf{value} & v & ::= & \lambda x : A.\, e \mid \Lambda \alpha.\, e \\ \textsf{typing context} & \Gamma & ::= & \cdot \mid \Gamma, x : A \mid \Gamma, \alpha\ \textsf{type} \end{array}$$

Note that a typing context Γ is an *ordered* set of type bindings and type declarations.

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

$$\begin{array}{rcl} e \mapsto e' & \Leftrightarrow & e \text{ reduces to } e' \\ \Gamma \vdash A\ \textsf{type} & \Leftrightarrow & A \text{ is a valid type with respect to typing context } \Gamma \\ \Gamma \vdash e : A & \Leftrightarrow & e \text{ has type } A \text{ under typing context } \Gamma \end{array}$$

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

$$\frac{e \mapsto e'}{e\, [\![A]\!] \mapsto e'\, [\![A]\!]}\ \textit{Tlam} \qquad \frac{}{(\Lambda \alpha.\, e)\, [\![A]\!] \mapsto [A/\alpha]e}\ \textit{Tapp}$$

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

$$\frac{\Gamma, \alpha\ \textsf{type} \vdash e : A}{\Gamma \vdash \Lambda \alpha.\, e : \forall \alpha. A}\ \forall\mathsf{I} \qquad \frac{\Gamma \vdash e : \forall \alpha. B \quad \Gamma \vdash A\ \textsf{type}}{\Gamma \vdash e\, [\![A]\!] : [A/\alpha]B}\ \forall\mathsf{E}$$

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 $\Gamma \vdash A$ type and Γ, α $\mathsf{type}, \Gamma' \vdash B$ type, then $\Gamma, [A/\alpha]\Gamma' \vdash [A/\alpha]B$ type.

(for substituting types for type variables in expressions)
If $\Gamma \vdash A$ type and Γ, α $\mathsf{type}, \Gamma' \vdash e : B$, 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 $A \times B$, pair, and fst in System F:

$$A \times B = \forall \alpha.(A \rightarrow B \rightarrow \alpha) \rightarrow \alpha$$

$$\mathsf{pair} \ : \ \forall \alpha.\forall \beta.\alpha \rightarrow \beta \rightarrow \alpha \times \beta \ = \ \Lambda \alpha.\, \Lambda \beta.\, \lambda x : \alpha.\, \lambda y : \beta.\, \Lambda \gamma.\, \lambda f : \alpha \rightarrow \beta \rightarrow \gamma.\, f\ x\ y$$

$$\mathsf{fst} \ : \ \forall \alpha.\forall \beta.\alpha \times \beta \rightarrow \alpha \ = \ \Lambda \alpha.\, \Lambda \beta.\, \lambda p : \alpha \times \beta.\, p\, [\![\alpha]\!]\ (\lambda x : \alpha.\, \lambda y : \beta.\, x)$$

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 $e\,[\![A]\!]$ ranges over polymorphic types.

7 Predicative polymorphic λ-calculus [4 pts]

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

$$
\begin{array}{rrcl}
\textsf{monotype} & A & ::= & A \to A \mid \alpha \\
\textsf{polytype} & U & ::= & A \mid \forall\alpha.U \\
\textsf{expression} & e & ::= & x \mid \lambda x{:}A.\,e \mid e\;e \mid \Lambda\alpha.\,e \mid e\,[\![A]\!] \\
\textsf{value} & v & ::= & \lambda x{:}A.\,e \mid \Lambda\alpha.\,e \\
\textsf{typing context} & \Gamma & ::= & \cdot \mid \Gamma, x : A \mid \Gamma, \alpha\ \textsf{type}
\end{array}
$$

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

$$
\frac{\Gamma, \alpha\ \textsf{type} \vdash e : U}{\Gamma \vdash \Lambda\alpha.\,e : \forall\alpha.U}\ \forall\mathsf{I}
\qquad
\frac{\Gamma \vdash e : \forall\alpha.U \quad \Gamma \vdash A\ \textsf{type}}{\Gamma \vdash e\,[\![A]\!] : [A/\alpha]U}\ \forall\mathsf{E}
$$

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.
- $\textsf{true}$ has type $\textsf{bool}$ in both System F and the predicative polymorphic λ-calculus.
- 0 has type $\textsf{int}$ in both System F and the predicative polymorphic λ-calculus.

$$
(\lambda f.\,((f\ \textsf{true}), (f\ 0)))\ (\lambda x.\,x)
$$

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:

  $$\text{untyped expression} \quad e \quad ::= \quad \cdots$$

- reduction judgment:

  $$e \to e' \quad \Leftrightarrow \quad e \textit{ reduces to } e'$$

2) typed language L_t

- syntax:

  $$\begin{array}{lll} \text{type} & A & ::= \quad \cdots \\ \text{typed expression} & t & ::= \quad \cdots \end{array}$$

- typing judgment:

  $$t : A \quad \Leftrightarrow \quad t \textit{ has type } A$$

- reduction judgment:

  $$t \Rightarrow t' \quad \Leftrightarrow \quad t \textit{ reduces to } t'$$

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 $e \to^* e'$

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 λ-calculus:

$$
\begin{array}{rrcl}
\textsf{monotype} & A & ::= & A \to A \mid \alpha \\
\textsf{polytype} & U & ::= & A \mid \forall\alpha.U \\
\textsf{expression} & e & ::= & x \mid \lambda x.\, e \mid e\ e \mid \textsf{let}\ x = e\ \textsf{in}\ e \\
\textsf{typing context} & \Gamma & ::= & \cdot \mid \Gamma, x : U \\
\textsf{type substitution} & S & ::= & \mathrm{id} \mid \{A/\alpha\} \mid S \circ S \\
\textsf{type equations} & E & ::= & \cdot \mid E, A = A
\end{array}
$$

We use the following auxiliary functions and notations:

- $S \cdot U$ and $S \cdot \Gamma$ denote applications of S to U and Γ, respectively.
- $\mathit{ftv}(\Gamma)$ denotes the set of free type variables in Γ; $\mathit{ftv}(U)$ 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 Γ contains no type binding for variable x.

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

$$
\frac{x : U \in \Gamma}{\Gamma \triangleright x : U}\ \textsf{Var}
\qquad
\frac{\Gamma, x : A \triangleright e : B}{\Gamma \triangleright \lambda x.\, e : A \to B}\ {\to}\textsf{I}
\qquad
\frac{\Gamma \triangleright e : A \to B \quad \Gamma \triangleright e' : A}{\Gamma \triangleright e\ e' : B}\ {\to}\textsf{E}
$$

$$
\frac{\Gamma \triangleright e : U \quad \Gamma, x : U \triangleright e' : A}{\Gamma \triangleright \textsf{let}\ x = e\ \textsf{in}\ e' : A}\ \textsf{Let}
\qquad
\frac{\Gamma \triangleright e : U \quad \alpha \notin \mathit{ftv}(\Gamma)}{\Gamma \triangleright e : \forall\alpha.U}\ \textsf{Gen}
\qquad
\frac{\Gamma \triangleright e : \forall\alpha.U}{\Gamma \triangleright e : [A/\alpha]U}\ \textsf{Spec}
$$

Question 1. [**3 pts**] $\mathsf{Unify}(E)$ 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, $\mathsf{Unify}(E)$ returns *fail*. Complete the definition of $\mathsf{Unify}(E)$.

$$\mathsf{Unify}(\cdot) \quad = \quad \text{id}$$

$$\mathsf{Unify}(E, \alpha = A) = \mathsf{Unify}(E, A = \alpha) \quad = \quad \text{if } \alpha = A \text{ then} \mathsf{Unify}(E)$$

$$\text{else if } \alpha \in \mathit{ftv}(A) \text{ then } \mathit{fail}$$

$$\text{else } \mathsf{Unify}(\{A/\alpha\} \cdot E) \circ \{A/\alpha\}$$

$$\mathsf{Unify}(E, A_1 \rightarrow A_2 = B_1 \rightarrow B_2) \quad = \quad \mathsf{Unify}(E, A_1 = B_1, A_2 = B_2)$$

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

$$\mathsf{Gen}_{\cdot}(\alpha \rightarrow \alpha) \quad = \quad \forall \alpha. \alpha \rightarrow \alpha$$

$$\mathsf{Gen}_{x:\alpha}(\alpha \rightarrow \alpha) \quad = \quad \alpha \rightarrow \alpha$$

$$\mathsf{Gen}_{x:\alpha}(\alpha \rightarrow \beta) \quad = \quad \forall \beta. \alpha \rightarrow \beta$$

$$\mathsf{Gen}_{x:\alpha, y:\beta}(\alpha \rightarrow \beta) \quad = \quad \alpha \rightarrow \beta$$

Question 3. [7 pts] The type reconstruction algorithm $\mathcal{W}$ takes a typing context Γ and an expression e as input, and returns a pair of a type substitution S and a monotype A as output:

$$\mathcal{W}(\Gamma, e) \;=\; (S, A)$$

Complete the definition of the algorithm $\mathcal{W}$:

$$\begin{array}{rcll}
\mathcal{W}(\Gamma, x) & = & (\mathrm{id}, \{\vec{\beta}/\vec{\alpha}\} \cdot A) & x : \forall\vec{\alpha}.A \in \Gamma \text{ and fresh } \vec{\beta} \\
\mathcal{W}(\Gamma, \lambda x.\, e) & = & \text{let } (S, A) = \mathcal{W}(\Gamma + x : \alpha, e) \text{ in} & \text{fresh } \alpha \\
 & & (S, (S \cdot \alpha) \rightarrow A) & \\
\mathcal{W}(\Gamma, e_1\ e_2) & = & \text{let } (S_1, A_1) = \mathcal{W}(\Gamma, e_1) \text{ in} & \\
 & & \text{let } (S_2, A_2) = \mathcal{W}(S_1 \cdot \Gamma, e_2) \text{ in} & \\
 & & \text{let } S_3 = \mathsf{Unify}(S_2 \cdot A_1 = A_2 \rightarrow \alpha) \text{ in} & \text{fresh } \alpha \\
 & & (S_3 \circ S_2 \circ S_1, S_3 \cdot \alpha) & \\
\mathcal{W}(\Gamma, \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2) & = & \text{let } (S_1, A_1) = \mathcal{W}(\Gamma, e_1) \text{ in} & \\
 & & \text{let } (S_2, A_2) = \mathcal{W}(S_1 \cdot \Gamma + x : \mathsf{Gen}_{S_1 \cdot \Gamma}(A_1), e_2) \text{ in} & \\
 & & (S_2 \circ S_1, A_2) &
\end{array}$$

Question 4. [2 pts] State the soundness theorem of the algorithm $\mathcal{W}$:

(Soundness of $\mathcal{W}$)
If $\mathcal{W}(\Gamma, e) = (S, A)$, then $S \cdot \Gamma \triangleright e : A$.

Work sheet