Name: Hemos ID:

CSE-321 Programming Languages 2011
Final — Sample Solution

	Prob 1	Prob 2	Prob 3	Prob 4	Prob 5	Prob 6	Total
Score							
Max	15	15	10	17	18	25	100

- There are six problems on 18 pages in this exam, including one extracredit question.
- 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.

Instructor-Thank-Students-Problem [Extracredit]

State "Yes" if you attended all the lectures in this course, without missing a single lecture.

____

`PL 2011`$^\lambda$ [Extracredit]

State "Yes" if you wear the `PL 2011`$^\lambda$ T-shirt. ________________________

1 De Bruijn levels [15 pts]

In class, we learned about de Bruijn indexes which eliminate names of variables in the λ-calculus and thus dispenses with α-conversion, i.e., renaming in terms. De Bruijn's original article actually contains two different notations for nameless representations of terms: de Bruijn indexes and de Bruijn levels.

Similarly to de Bruijn indexes, de Bruijn levels also use integers, called "levels," to represent variables. The key difference is that de Bruijn levels number binders "from the outside in," whereas de Bruijn indexes number binders "from the inside out." For example, expression $\lambda x.\,(\lambda y.\,\lambda z.\,x\ y\ z)\ (\lambda w.\,w)$ is represented using de Bruijn indexes and de Bruijn levels as follows:

$$\begin{array}{lcll} \lambda x.\,(\lambda y.\,\lambda z.\,x\ y\ z)\ (\lambda w.\,x\ w) & \equiv_{\mathsf{dB_I}} & \lambda.\,(\lambda.\,\lambda.\,2\ 1\ 0)\ (\lambda.\,1\ 0) & \text{(using de Bruijn indexes)} \\ \lambda x.\,(\lambda y.\,\lambda z.\,x\ y\ z)\ (\lambda w.\,x\ w) & \equiv_{\mathsf{dB_L}} & \lambda.\,(\lambda.\,\lambda.\,0\ 1\ 2)\ (\lambda.\,0\ 1) & \text{(using de Bruijn levels)} \end{array}$$

Recall that a de Bruijn index for a variable specifies the relative position of its corresponding λ-binder. This, in turn, implies that the same variable can be assigned different de Bruijn indexes depending on its position. In contrast, *the* de Bruijn level for a variable specifies the position of its corresponding λ-binder relative to the outermost λ-binder. Hence the same variable is assigned a unique de Bruijn level regardless of its position inside an expression. In the example above, the two instances of the same variable x are assigned the same de Bruijn level 1.

To exploit de Bruijn levels in implementing the operational semantics of the λ-calculus, we introduce a substitution function $\delta_{i,k}(M, N)$ which substitutes N for a variable in M. We define $\delta_{i,k}(M, N)$ so that the following relationship holds; the meaning of subscripts i and k is explained later:

$$\begin{array}{ccc} (\lambda x.\,e)\ e' & \mapsto & [e'/x]e \\ \equiv_{\mathsf{dB_L}} & & \equiv_{\mathsf{dB_L}} \\ (\lambda.\,M)\ N & \mapsto & \delta_{i,k}(M, N) \end{array}$$

That is, applying $\lambda.\,M$ to N, or substituting N for the outermost variable bound in $\lambda.\,M$, results in $\delta_{i,k}(M, N)$. Here is a simple example:

$$\begin{array}{ccccc} (\lambda x.\,x)\ (\lambda y.\,y) & \mapsto & [(\lambda y.\,y)/x]x & = & \lambda y.\,y \\ \equiv_{\mathsf{dB_L}} & & \equiv_{\mathsf{dB_L}} & & \equiv_{\mathsf{dB_L}} \\ (\lambda.\,0)\ (\lambda.\,0) & \mapsto & \delta_{0,0}(0, \lambda.\,0) & = & \lambda.\,0 \end{array}$$

In $\delta_{i,k}(M, N)$, the subscript i is the de Bruijn level of the variable to be replaced by N. That is, we are to substitute N for de Bruijn level i in M. The subscript k denotes the number of λ-binders encountered so far, *i.e.*, before reaching M. That is, it denotes the number of λ-binders lying between the λ-binder corresponding to the de Bruijn level i and M. Consider the following example in which the redex is underlined:

$$\lambda x.\,\underline{(\lambda y.\,\lambda z.\,y)\ (\lambda w.\,x\ w)} \quad \equiv_{\mathsf{dB_L}} \quad \lambda.\,\underline{(\lambda.\,\lambda.\,1)\ (\lambda.\,0\ 1)} \quad \mapsto \quad \lambda.\,\delta_{1,0}(\lambda.\,1, \lambda.\,0\ 1)$$

Here the first subscript 1 is the de Bruijn level of variable y, which is to be replaced in this substitution. The second subscript is 0 because there is no intervening λ-binder between λy and $\lambda z.\,y$. We propagate the substitution function δ, its second subscript changes to 1 because the λ-binder λz:

$$\lambda.\,\delta_{1,0}(\lambda.\,1, \lambda.\,0\ 1) \quad = \quad \lambda.\,\lambda.\,\delta_{1,1}(1, \lambda.\,0\ 1)$$

The goal in this problem is to complete the definition of the substitution function $\delta_{i,k}(M, N)$. As with de Bruijn indexes, you will need a shifting function $\tau_i^k(N)$ which shifts by k all de Bruijn levels in N that are greater than or equal to i.

Question 1. [5 pts] Fill in the blank with appropriate variable names so that the following relation holds:

$$\lambda \underline{x}.\, \lambda \underline{y}.\, (\lambda \underline{z}.\, (\lambda \underline{u}.\, u\ z)\ z)\ (\lambda \underline{w}.\, x\ y\ w) \quad \equiv_{\mathsf{dB_L}} \quad \lambda.\, \lambda.\, (\lambda.\, (\lambda.\, 3\ 2)\ 2)\ (\lambda.\, 0\ 1\ 2)$$

Question 2. [5 pts] Complete the definition of $\tau_i^k(N)$.

$$\tau_i^k(N_1\ N_2) \quad = \quad \underline{\tau_i^k(N_1)\ \tau_i^k(N_2)}$$

$$\tau_i^k(\lambda.\, N) \quad = \quad \underline{\lambda.\, \tau_i^k(N)}$$

$$\tau_i^k(n) \quad = \quad \underline{n + k} \qquad \text{if } n \geq i$$

$$\tau_i^k(n) \quad = \quad \underline{n} \qquad \text{if } n < i$$

Question 3. [5 pts] Use $\tau_i^k(N)$ and complete the definition of $\delta_{i,k}(M, N)$.

$$\delta_{i,k}(M_1\ M_2, N) \quad = \quad \underline{\delta_{i,k}(M_1, N)\ \delta_{i,k}(M_2, N)}$$

$$\delta_{i,k}(\lambda.\, M, N) \quad = \quad \underline{\lambda.\, \delta_{i,k+1}(M, N)}$$

$$\delta_{i,k}(m, N) \quad = \quad \underline{m} \qquad \text{if } \ m < i$$

$$\delta_{i,k}(i, N) \quad = \quad \underline{\tau_i^k(N)}$$

$$\delta_{i,k}(m, N) \quad = \quad \underline{m - 1} \qquad \text{if } \ m > i$$

2 Mutable references [15 pts]

Question 1. [**8 pts**] Consider the simply-typed λ-calculus extended with mutable references; we assume syntactic sugar $\mathsf{let}\ x = e\ \mathsf{in}\ e'$ for $(\lambda x\!:\!A.\,e')\ e$ for some type A and various constructs for base types bool and nat:

$$\begin{array}{rrcl}
\mathsf{type} & A & ::= & P \mid A\to A \mid A+A \mid \mathsf{unit} \mid \mathsf{ref}\ A \mid \mathsf{bool} \mid \mathsf{nat} \\
\mathsf{expression} & e & ::= & x \mid \lambda x\!:\!A.\,e \mid e\ e \mid \mathsf{inl}_A\ e \mid \mathsf{inr}_A\ e \mid \mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}\ x.\,e \mid \mathsf{inr}\ x.\,e \mid () \mid \\
 & & & \mathsf{ref}\ e \mid\ !e \mid e := e \mid \\
 & & & \mathsf{let}\ x = e\ \mathsf{in}\ e \mid \\
 & & & \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid\ =\ \mid\ <\ \mid 0 \mid 1 \mid \cdots
\end{array}$$

In class, we learned how to simulate infinite arrays with functions. In this problem, we simulate *finite* arrays of natural numbers with functions. We choose a functional representation of arrays by defining type narray for finite arrays of natural numbers as follows:

$$\mathsf{narray} = \mathsf{ref}\ (\mathsf{nat}\to(\mathsf{unit}+\mathsf{nat}))$$

That is, we represent an array with a function of type $\mathsf{nat}\to\mathsf{unit}+\mathsf{nat}$ taking an index i of type nat. Depending on whether i is less than the size of the array or not, the function returns either a unit value or a corresponding element of the array.

We need the following constructs for finite arrays of natural numbers:

- $\mathsf{new} : \mathsf{nat}\to\mathsf{narray}$ for creating a new array.
  $\mathsf{new}\ n$ creates a new array of size n whose elements are initialized as 0.
- $\mathsf{access} : \mathsf{narray}\to\mathsf{nat}\to(\mathsf{unit}+\mathsf{nat})$ for accessing an array.
  $\mathsf{access}\ a\ i$ returns $\mathsf{inr}_{\mathsf{unit}}\ v$ where v is the i-th element of array a. If i is out of bound, *i.e.*, i greater than or equal to the size of the array, it returns $\mathsf{inl}_{\mathsf{nat}}\ ()$.
- $\mathsf{update} : \mathsf{narray}\to\mathsf{nat}\to\mathsf{nat}\to\mathsf{unit}$ for updating an array.
  $\mathsf{update}\ a\ i\ v$ updates the i-th element of array a with integer v if i is less than the size of the array. It i is out of bounds, it leaves the array intact.

Implement new, access, and update:

$$\mathsf{new} = \lambda b\!:\!\mathsf{nat}.\,\mathsf{ref}\ \lambda i\!:\!\mathsf{nat}.\,\mathsf{if}\ i < b\ \mathsf{then}\ \mathsf{inr}_{\mathsf{unit}}\ 0\ \mathsf{else}\ \mathsf{inl}_{\mathsf{nat}}\ ()$$

$$\mathsf{access} = \lambda a\!:\!\mathsf{iarray}.\,\lambda i\!:\!\mathsf{nat}.\,(!a)\ i$$

$$\mathsf{update} = \lambda a\!:\!\mathsf{iarray}.\,\lambda i\!:\!\mathsf{nat}.\,\lambda n\!:\!\mathsf{nat}.$$

$$\mathsf{case}\ (!a)\ i\ \mathsf{of}\ \mathsf{inl}\ \_.\ () \mid \mathsf{inr}\ \_.\ \mathsf{let}\ \mathit{old} = !a\ \mathsf{in}\ a := \lambda j\!:\!\mathsf{int}.\,\mathsf{if}\ i = j\ \mathsf{then}\ \mathsf{inr}_{\mathsf{unit}}\ n\ \mathsf{else}\ \mathit{old}\ j$$

Question 2. [7 pts] Consider the following simply-typed λ-calculus extended with mutable references:

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

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

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

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 $dom(\psi)$:

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

We write $\Psi(l)$ for the type to which l is mapped under Ψ; in order for $\Psi(l)$ to be defined, l must be in $dom(\Psi)$:

$$(\Psi', l \mapsto A)(l) \quad = \quad A$$

We use the following reduction judgment which carries a store along with an expression being reduced:

$$e \mid \psi \mapsto e' \mid \psi' \qquad \Leftrightarrow \qquad \textit{e with store } \psi \textit{ reduces to } e' \textit{ with store } \psi'$$

We also use the following typing judgment:

$$\Gamma \mid \Psi \vdash e : A \quad \Leftrightarrow \quad \textit{expression } e \textit{ has type } A \textit{ under typing context } \Gamma \textit{ and store typing context } \Psi$$

Complete reduction rules and typing rules shown below:

(Reduction rules)

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

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

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

(Typing rules)

$$\frac{\Psi(l) = A}{\Gamma \mid \Psi \vdash l : \mathsf{ref}\ A}\ \mathsf{Loc}$$

$$\frac{\Gamma \mid \Psi \vdash e : A}{\Gamma \mid \Psi \vdash \mathsf{ref}\ e : \mathsf{ref}\ A}\ \mathsf{Ref}$$

$$\frac{\Gamma \mid \Psi \vdash e : \mathsf{ref}\ A}{\Gamma \mid \Psi \vdash\ !e : A}\ \mathsf{Deref}$$

$$\frac{\Gamma \mid \Psi \vdash e : \mathsf{ref}\ A \quad \Gamma \mid \Psi \vdash e' : A}{\Gamma \mid \Psi \vdash e := e' : \mathsf{unit}}\ \mathsf{Assign}$$

3 Subtyping [10 pts]

Consider the following definition:

$$\begin{array}{rrcl} \textsf{type} & A & ::= & P \mid A \to A \mid A \times A \mid A + A \mid \textsf{ref}\ A \\ \textsf{base type} & P & ::= & \textsf{nat} \mid \textsf{int} \end{array}$$

We write $A \leq B$ if A is a subtype of B, or equivalently, if B is a *supertype* of A. Complete the following subtyping rules for product types, function types, and reference types:

$$\frac{A \leq A' \quad B \leq B'}{A \times B \leq A' \times B'}\ \mathit{Prod}_{\leq}$$

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

$$\frac{A \leq B \quad B \leq A}{\textsf{ref}\ A \leq \textsf{ref}\ B}\ \mathit{Ref}_{\leq}$$

4 Recursive types [17 pts]

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

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

Question 1. [4 pts] Write the reduction rules for constructs $\mathsf{fold}_C\ e$ and $\mathsf{unfold}_C\ e$ under the eager reduction strategy:

$$\frac{e \mapsto e'}{\mathsf{fold}_C\ e \mapsto \mathsf{fold}_C\ e'}\ \mathit{Fold}$$

$$\frac{e \mapsto e'}{\mathsf{unfold}_C\ e \mapsto \mathsf{unfold}_C\ e'}\ \mathit{Unfold}$$

$$\frac{}{\mathsf{unfold}_C\ \mathsf{fold}_C\ v \mapsto v}\ \mathit{Unfold}^2$$

Question 2. [3 pts] State the reason why we do not need a reduction rule for $\mathsf{fold}_C\ \mathsf{unfold}_C\ v$ (where v is a value):

If $v = \mathsf{fold}_C\ v'$ for a value v', then

$$\mathsf{fold}_C\ \mathsf{unfold}_C\ v = \mathsf{fold}_C\ \mathsf{unfold}_C\ \mathsf{fold}_C\ v' \mapsto \mathsf{fold}_C\ v'$$

Otherwise, $\mathsf{fold}_C\ \mathsf{unfold}_C\ v$ get stuck. Therefore, we do not need any rule for $\mathsf{fold}_C\ \mathsf{unfold}_C\ v$.

Question 3. [**5 pts**] Translate the following definition in SML for lists of natural numbers into the simply-typed λ-calculus with recursive types.

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

Question 4. [**2 pts**] In this question, we define a recursive type `hungry`. When unfolded, an expression of type `hungry` yields a function of type $\mathsf{nat}\rightarrow\texttt{hungry}$, which takes an integer and returns another expression of type `hungry`. For example, if f is of type `hungry`, then $\mathsf{unfold}_{\texttt{hungry}}\ f\ 0$ returns an expression of type `hungry`, say f_0. Since f_0 is also of type `hungry`, it is possible to apply $\mathsf{unfold}_{\texttt{hungry}}\ f_0$ to any integer. An expression of type `hungry` can be unfolded and applied to an integer infinitely many times, so we call it a "hungry" function.

Define the recursive type `hungry`:

$$\texttt{hungry} = \mu\alpha.\mathsf{nat}\rightarrow\alpha$$

Question 5. [**3 pts**] Give an expression e of type `hungry` such that the evaluation of $\mathsf{unfold}_e\ \texttt{hungry}\ n$ terminates for any natural number n of type nat.

$$\mathsf{fold}_{\texttt{hungry}}\ \mathsf{fix}\ f:\mathsf{nat}\rightarrow\texttt{hungry}.\ \lambda i:\mathsf{int}.\ \mathsf{fold}_{\texttt{hungry}}\ f$$

5 Polymorphism [18 pts]

Question 1. [**5 pts**] In this question, we encode sum types and its related constructs, inl, inr, and case, in System F. A sum type A_1+A_2 contains a value of type A_1 or a value of type A_2, but not both. inl takes a type and an expression, and injects the expression to the left of a sum type; inr injects a given expression to the right of a sum type. case performs a case analysis on an expression of a sum type.

We encode the sum type $A+B$ in System F as follows:

$$A+B = \forall\alpha.(A\to\alpha)\to(B\to\alpha)\to\alpha$$

Informally the idea is that a sum type $A+B$ allows us to produce a value of a certain type, say α, whenever we know how to produce a value of type α from either type A or type B.

Encode inl, inr, and case in System F. We provide the type of each construct. For your reference, we also provide the syntax for System F:

$$\begin{array}{rrcl} \mathsf{type} & A & ::= & A\to A \mid \alpha \mid \forall\alpha.A \\ \mathsf{expression} & e & ::= & x \mid \lambda x\!:\!A.\,e \mid e\ e \mid \Lambda\alpha.\,e \mid e\,[\![A]\!] \end{array}$$

$$\begin{array}{rcll} \mathsf{inl} & : & \forall\alpha.\forall\beta.\alpha\to\alpha+\beta & = \Lambda\alpha.\,\Lambda\beta.\,\lambda x\!:\!\alpha.\,\Lambda\gamma.\,\lambda f\!:\!\alpha\to\gamma.\,\lambda g\!:\!\beta\to\gamma.\,f\ x \\ \mathsf{inr} & : & \forall\alpha.\forall\beta.\beta\to\alpha+\beta & = \Lambda\alpha.\,\Lambda\beta.\,\lambda y\!:\!\beta.\,\Lambda\gamma.\,\lambda f\!:\!\alpha\to\gamma.\,\lambda g\!:\!\beta\to\gamma.\,g\ y \\ \mathsf{case} & : & \forall\alpha.\forall\beta.\forall\gamma.\alpha+\beta\to(\alpha\to\gamma)\to(\beta\to\gamma)\to\gamma & = \Lambda\alpha.\,\Lambda\beta.\,\Lambda\gamma.\,\lambda s\!:\!\alpha+\beta.\,\lambda f\!:\!\alpha\to\gamma.\,\lambda g\!:\!\beta\to\gamma.\,s\,[\![\gamma]\!]\ f\ g \end{array}$$

Question 2. [**2 pts**] Is $(\lambda f.\,\mathsf{pair}\ (f\ \mathsf{true})\ (f\ 0))\ (\lambda x.\,x)$ typable in the <u>predicative</u> polymorphic λ-calculus? State “Yes” or “No.”

No

Question 3. [**2 pts**] Is $(\lambda f.\,\mathsf{pair}\ (f\ \mathsf{true})\ (f\ 0))\ (\lambda x.\,x)$ typable in the <u>let-polymorphic</u> λ-calculus? State “Yes” or “No.”

No

Question 4. [3 pts] Consider the implicit let-polymorphic λ-calculus:

$$\begin{array}{rrcl}
\textsf{monotype} & A & ::= & A \rightarrow A \mid \alpha \\
\textsf{polytype} & U & ::= & A \mid \forall \alpha . U \\
\textsf{expression} & e & ::= & x \mid \lambda x.\, e \mid e\ e \mid \textsf{let}\ x = e\ \textsf{in}\ e \\
\textsf{value} & v & ::= & \lambda x.\, e \\
\textsf{typing context} & \Gamma & ::= & \cdot \mid \Gamma, x : U \mid \Gamma, \alpha\ \textsf{type}
\end{array}$$

We use a typing judgment $\Gamma \triangleright e : U$ to express that untyped expression e is typable with a polytype U. Complete the following typing rules (generalization and specialization):

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

$$\frac{\Gamma \triangleright e : A \rightarrow B \quad \Gamma \triangleright e' : A}{\Gamma \triangleright e\ e' : B}\ {\rightarrow}\textsf{E} \qquad \frac{\Gamma \triangleright e : U \quad \Gamma, x : U \triangleright e' : A}{\Gamma \triangleright \textsf{let}\ x = e\ \textsf{in}\ e' : A}\ \textsf{Let}$$

$$\frac{\Gamma, \alpha\ \textsf{type} \triangleright e : U}{\Gamma \triangleright e : \forall \alpha . U}\ \textsf{Gen} \qquad \frac{\Gamma \triangleright e : \forall \alpha . U \quad \Gamma \vdash A\ \textsf{type}}{\Gamma \triangleright e : [A/\alpha]U}\ \textsf{Spec}$$

Question 5. [3 pts] Complete the typing derivation for expression $\lambda x.\, \lambda y.\, (x, y)$ in the implicit let-polymorphic type system from Question 4:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, \alpha\ \textsf{type}, \beta\ \textsf{type}, x : \alpha, y : \beta \triangleright x : \alpha}\ \textsf{Var} \quad \dfrac{}{\Gamma, \alpha\ \textsf{type}, \beta\ \textsf{type}, x : \alpha, y : \beta \triangleright y : \beta}\ \textsf{Var}}{\Gamma, \alpha\ \textsf{type}, \beta\ \textsf{type}, x : \alpha, y : \beta \triangleright (x, y) : (\alpha \times \beta)}\ {\times}\textsf{I}}{\Gamma, \alpha\ \textsf{type}, \beta\ \textsf{type}, x : \alpha \triangleright \lambda y.\, (x, y) : \beta \rightarrow (\alpha \times \beta)}\ {\rightarrow}\textsf{I}}{\Gamma, \alpha\ \textsf{type}, \beta\ \textsf{type} \triangleright \lambda x.\, \lambda y.\, (x, y) : \alpha \rightarrow \beta \rightarrow (\alpha \times \beta)}\ {\rightarrow}\textsf{I}}{\Gamma, \alpha\ \textsf{type} \triangleright \lambda x.\, \lambda y.\, (x, y) : \forall \beta . \alpha \rightarrow \beta \rightarrow (\alpha \times \beta)}\ \textsf{Gen}}{\Gamma \triangleright \lambda x.\, \lambda y.\, (x, y) : \forall \alpha . \forall \beta . \alpha \rightarrow \beta \rightarrow (\alpha \times \beta)}\ \textsf{Gen}$$

Question 6. [3 pts] Modify the rule Gen in Question 4 so that the type system satisfies value restriction:

$$\frac{\Gamma, \alpha\ \textsf{type} \triangleright v : U}{\Gamma \triangleright v : \forall \alpha . U}\ \textsf{Gen}$$

6 Type reconstruction [25 pts]

In this problem, we examine the type reconstruction algorithm $\mathcal{W}$ for the implicitly let-polymorphic λ-calculus:

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

A type substitution is a mapping from type variables to monotypes. Two auxiliary functions $ftv(A)$ and $ftv(\Gamma)$ denote the set of free type variables in A and Γ, respectively:

$$\begin{array}{rcl}
ftv(\cdot) & = & \varnothing \\
ftv(\Gamma, x : U) & = & ftv(\Gamma) \cup ftv(U)
\end{array}
\qquad
\begin{array}{rcl}
ftv(A \to B) & = & ftv(A) \cup ftv(B) \\
ftv(\alpha) & = & \{\alpha\} \\
ftv(\forall\alpha.U) & = & ftv(U) - \{\alpha\}
\end{array}$$

Typing rules are:

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

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

Question 1. [3 pts] Complete the definition of the application of a type substitution on types:

$$\begin{array}{rcll}
\mathrm{id} \cdot U & = & U & \\
\{A/\alpha\} \cdot \alpha & = & A & \\
\{A/\alpha\} \cdot \beta & = & \beta & \text{where } \alpha \neq \beta \\
\{A/\alpha\} \cdot B_1 \to B_2 & = & \{A/\alpha\}B_1 \to \{A/\alpha\}B_2 & \\
\{A/\alpha\} \cdot \forall\alpha.U & = & \forall\alpha.U & \\
\{A/\alpha\} \cdot \forall\beta.U & = & \forall\beta.\{A/\alpha\}U & \text{where } \alpha \neq \beta \\
S_1 \circ S_2 \cdot U & = & S_1 \cdot (S_2 \cdot U) &
\end{array}$$

Question 2. [**5 pts**] $\mathsf{Unify}(E)$ is a function which 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 $\mathsf{Unify}(E)$.

$$\begin{array}{rcl}
\mathsf{Unify}(\cdot) & = & \mathrm{id} \\
\mathsf{Unify}(E, \alpha = A) = \mathsf{Unify}(E, A = \alpha) & = & \text{if } \alpha = A \text{ then} \mathsf{Unify}(E) \\
 & & \text{else if } \alpha \in \mathit{ftv}(A) \text{ then } \mathit{fail} \\
 & & \text{else } \mathsf{Unify}(\{A/\alpha\} \cdot E) \circ \{A/\alpha\} \\
\mathsf{Unify}(E, A_1 \to A_2 = B_1 \to B_2) & = & \mathsf{Unify}(E, A_1 = B_1, A_2 = B_2)
\end{array}$$

Question 3. [**3 pts**] Complete the function $\mathsf{Gen}_\Gamma(A)$ which generalizes monotype A to a polytype after taking into account free type variables in typing context Γ:

$$\mathsf{Gen}_\Gamma(A) \; = \; \forall \alpha_1 . \forall \alpha_2 . \cdots \forall \alpha_n . A \text{ where } \alpha_i \notin \mathit{ftv}(\Gamma) \text{ and } \alpha_i \in \mathit{ftv}(A) \text{ for } i = 1, \cdots, n.$$

Question 4. [14 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:

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

$S \cdot \Gamma$ applies type substitution S to Γ:

$$S \cdot (\Gamma, x : U) \;=\; S \cdot \Gamma, x : (S \cdot U)$$

Complete the definition of 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 5. **[Extracredit]** Prove the soundness of the algorithm $\mathcal{W}$. You may use the following three lemmas with proofs:

Lemma A. If $\Gamma \triangleright e : A$, then $S \cdot \Gamma \triangleright e : S \cdot A$ for any type substitution S.

Lemma B. If $\mathsf{Unify}(A_1 = A'_1, \cdots, A_n = A'_n) = S$, then $S \cdot A_i = S \cdot A'_i$ for $i = 1, \cdots, n$.

Lemma C. If $\Gamma \triangleright e : A$, then $\Gamma \triangleright e : \mathsf{Gen}_\Gamma(A)$.

Fill in the blank to complete the proof.

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

Proof. By structural induction on e. We consider three cases.

Case $e = \lambda x.\, e'$:

There exist S_1, A_1, and a fresh type variable α_1 such that:
$\mathcal{W}(\Gamma + x : \alpha_1, e') = (S_1, A_1)$ by the definition of $\mathcal{W}$

$S_1 \cdot \Gamma + x : S_1 \cdot \alpha_1 \triangleright e' : A_1$ by induction hypothesis on e'

$$\left.\begin{array}{l} \mathcal{W}(\Gamma, \lambda x.\, e') = (S_1, (S_1 \cdot \alpha_1) \to A_1) \\ S = S_1 \\ A = (S_1 \cdot \alpha_1) \to A_1 \end{array}\right\} \quad \text{by the definition of } \mathcal{W}$$

$S_1 \cdot \Gamma \triangleright \lambda x.\, e' : (S_1 \cdot \alpha_1) \to A_1$ by the rule $\to\mathsf{I}$ with $S_1 \cdot \Gamma + x : S_1 \cdot \alpha_1 \triangleright e' : A_1$

$S \cdot \Gamma \triangleright \lambda x.\, e' : A$ from $S = S_1$, $A = (S_1 \cdot \alpha_1) \to A_1$

Case $e = e_1\ e_2$:

There exist S_1 and A_1 such that
$\mathcal{W}(\Gamma, e_1) = (S_1, A_1)$ by the definition of $\mathcal{W}$

$S_1 \cdot \Gamma \triangleright e_1 : A_1$ by induction hypothesis on e_1

There exist S_2 and A_2 such that
$\mathcal{W}(S_1 \cdot \Gamma, e_2) = (S_2, A_2)$ by the definition of $\mathcal{W}$

$S_2 \cdot S_1 \cdot \Gamma \triangleright e_2 : A_2$ by induction hypothesis on e_2

There exists S_3 and *fresh* α such that
$S_3 = \mathsf{Unify}(S_2 \cdot A_1 = A_2 \to \alpha)$ by the definition of $\mathcal{W}$

$\mathcal{W}(\Gamma, e_1\ e_2) = (S_3 \circ S_2 \circ S_1, S_3 \cdot \alpha)$,
$S = S_3 \circ S_2 \circ S_1$,
$A = S_3 \cdot \alpha$ by the definition of $\mathcal{W}$

$S_3 \cdot S_2 \cdot S_1 \cdot \Gamma \rhd e_1 : S_3 \cdot S_2 \cdot A_1$ by Lemma **??** with $S_1 \cdot \Gamma \rhd e_1 : A_1$

$S_3 \cdot S_2 \cdot S_1 \cdot \Gamma \rhd e_2 : S_3 \cdot A_2$ by Lemma **??** with $S_2 \cdot S_1 \cdot \Gamma \rhd e_2 : A_2$

$S_3 \cdot S_2 \cdot A_1 = S_3 \cdot A_2 \rightarrow S_3 \cdot \alpha$ by Lemma **??** with $S_3 = \mathsf{Unify}(S_2 \cdot A_1 = A_2 \rightarrow \alpha)$

$S_3 \cdot S_2 \cdot S_1 \cdot \Gamma \rhd e_1 : S_3 \cdot A_2 \rightarrow S_3 \cdot \alpha$ from $S_3 \cdot S_2 \cdot A_1 = S_3 \cdot A_2 \rightarrow S_3 \cdot \alpha$

$S_3 \cdot S_2 \cdot S_1 \cdot \Gamma \rhd e_1\ e_2 : S_3 \cdot \alpha$ by the rule $\rightarrow\mathsf{E}$

$S \cdot \Gamma \rhd e_1\ e_2 : A$ from $S = S_3 \circ S_2 \circ S_1$ and $A = S_3 \cdot \alpha$

Case $e = \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$:

There exist S_1 and A_1 such that
$\mathcal{W}(\Gamma, e_1) = (S_1, A_1)$ by the definition of $\mathcal{W}$

$S_1 \cdot \Gamma \triangleright e_1 : A_1$ by induction hypothesis on e_1

There exist S_2 and A_2 such that
$\mathcal{W}(S_1 \cdot \Gamma + x : \mathsf{Gen}_{S_1 \cdot \Gamma}(A_1), e_2) = (S_2, A_2)$ by the definition of $\mathcal{W}$

$S_2 \cdot S_1 \cdot \Gamma + x : S_2 \cdot \mathsf{Gen}_{S_1 \cdot \Gamma}(A_1) \triangleright e_2 : A_2$ by induction hypothesis on e_2

$\mathcal{W}(\Gamma, \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2) = (S_2 \circ S_1, A_2)$,
$S = S_2 \circ S_1$,
$A = A_2$ by the definition of $\mathcal{W}$

$S_1 \cdot \Gamma \triangleright e_1 : \mathsf{Gen}_{S_1 \cdot \Gamma}(A_1)$ by Lemma **??** with $S_1 \cdot \Gamma \triangleright e_1 : A_1$

$S_2 \cdot S_1 \cdot \Gamma \triangleright e_1 : S_2 \cdot \mathsf{Gen}_{S_1 \cdot \Gamma}(A_1)$ by Lemma **??** with $S_1 \cdot \Gamma \triangleright e_1 : \mathsf{Gen}_{S_1 \cdot \Gamma}(A_1)$

$S_2 \cdot S_1 \cdot \Gamma \triangleright \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : A_2$ by the rule Let

$S \cdot \Gamma \triangleright \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : A$ from $S = S_2 \circ S_1$ and $A = A_2$
$\square$

Work sheet