Name: Hemos ID:

CSE-321 Programming Languages 2007
Final — Sample Solution

	Prob 1	Prob 2	Prob 3	Prob 4	Prob 5	Prob 6	Prob 7
Score							
Max	25	8	17	13	10	20	10
	Prob 8	Prob 9	Prob 10	Prob 11	Prob 12	Total	
Score							
Max	25	15	20	15	22	200 = 125 + 75	

- There are 12 problems on 20 pages, including a work sheet, in this exam.
- The maximum score for this exam is 200 points, 75 points of which are counted as extra-credit.
- Be sure to write your name and Hemos ID.
- You have three hours for this exam.

Good Boy, Good Girl [Extracredit]

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

Yes

1 Type safety and object-orientation [25 pts]

Question 1. [**5 pts**] For the simply typed λ-calculus, type safety consists of two theorems: progress and type preservation. Fill in the blank and complete the theorems:

Theorem (Progress). *If* $\cdot \vdash e : A$ *for some type* A*, then either* $\underline{\quad e \text{ is a value} \quad}$

or there exists $\underline{\quad e' \quad}$ *such that* $\underline{\quad e \mapsto e' \quad}$.

Theorem (Type preservation). *If* $\Gamma \vdash e : A$ *and* $\underline{\quad e \mapsto e' \quad}$*, then*

$\underline{\quad \Gamma \vdash e' : A \quad}$.

Question 2. [**7 pts**] For the simply typed λ-calculus extended with references, type safety also consists of two theorems: progress and type preservation. Fill in the blank and complete the theorems:

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

(1) $\underline{\quad e \text{ is a value} \quad}$*, or*

(2) for any $\underline{\quad \text{store } \psi \quad}$ *such that* $\underline{\quad \psi :: \Psi \quad}$*,*

there exist some $\underline{\quad \text{expression } e' \quad}$ *and* $\underline{\quad \text{store } \psi' \quad}$ *such that* $\underline{\quad e \mid \psi \mapsto e' \mid \psi' \quad}$.

Theorem (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* $\begin{cases} \underline{\quad \Gamma \mid \Psi' \vdash e' : A \quad} \\ \underline{\quad \Psi \subset \Psi' \quad} \\ \underline{\quad \psi' :: \Psi' \quad} \end{cases}$.

Question 3. **[5 pts]** True or false:

(1) *"If an expression is well-typed, it successfully evaluates to a value under the call-by-name strategy."* If true, state "true." If false, state "false" and give a counter-example that is well-typed, but fails to evaluate to a value. You may use any construct given in the Course Notes.

True

(2) *"If an expression successfully evaluates to a value under the call-by-name strategy, it is well-typed."* If true, state "true." If false, state "false" and give a counter-example that is ill-typed, but evaluates to a value successfully. You may use any construct given in the Course Notes.

False : $\mathsf{fst}\ (0, \mathsf{plus}(0, \mathsf{false}))$

Question 4. **[8 pts]** List the five key features of object-oriented languages that we discussed in class:

1. Multiple representations
2. Encapsulation
3. Subtyping
4. Inheritance
5. Open recursion

2 Mutable References [8 pts]

We want to represent an array of integers as a function taking an index (of type int) and returning a corresponding element of the array. We choose a functional representation of arrays by defining type iarray for arrays of integers as follows:

$$\mathsf{iarray} = \mathsf{ref}\ (\mathsf{int} \to \mathsf{int})$$

We need the following constructs for arrays:

- new : unit → iarray for creating a new array.
  new () returns a new array of indefinite size; all elements are initialized as 0.
- access : iarray → int → int for accessing an array.
  access a i returns the i-th element of array a.
- update : iarray → int → int → unit for updating an array.
  update a i n updates the i-the element of array a with integer n.

Exploit the constructs for mutable references to implement new, access, and update. Fill in the blank:

$$\mathsf{new} = \lambda\_{:}\mathsf{unit}.\ \underline{\mathsf{ref}\ \lambda i{:}\mathsf{int}.\ 0}$$

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

$$\mathsf{update} = \lambda a{:}\mathsf{iarray}.\ \lambda i{:}\mathsf{int}.\ \lambda n{:}\mathsf{int}.$$

$$\mathsf{let}\ \underline{old} = \underline{!a}\ \mathsf{in}$$

$$\underline{a := \lambda j{:}\mathsf{int}.\ \mathsf{if}\ i = j\ \mathsf{then}\ n\ \mathsf{else}\ old\ j}$$

3 Abstract Machine E [17 pts]

We want to develop an abstract machine E which is based on a reduction judgment (derived from the environment evaluation judgment) and makes no use of substitutions. Consider the fragment of the simply typed λ-calculus (where base types are unspecified).

$$\begin{array}{rrcl} \textsf{type} & A & ::= & P \mid A \to A \\ \textsf{base type} & P & & \\ \textsf{expression} & e & ::= & x \mid \lambda x{:}A.\, e \mid e\ e \mid \\ \textsf{value} & v & ::= & \lambda x{:}A.\, e \end{array}$$

Question 1. **[3 pts]** Complete the following definitions for the abstract machine E. Fill in the blank:

$$\begin{array}{rrcl} \textsf{value} & v & ::= & [\eta, \lambda x{:}A.\, e] \\ \textsf{environment} & \eta & ::= & \cdot \mid \eta, x \hookrightarrow v \\ \textsf{frame} & \phi & ::= & \square_\eta\ e \mid [\eta, \lambda x{:}A.\, e]\ \square \\ \textsf{stack} & \sigma & ::= & \square \mid \sigma; \phi \\ \textsf{state} & s & ::= & \sigma \blacktriangleright e \mathbin{@} \eta \mid \sigma \blacktriangleleft v \end{array}$$

Question 2. **[4 pts]** Complete the transition rules for the abstract machine E. You will need exactly five rules. We give a rule Var_{E}.

$$\frac{x \hookrightarrow v \in \eta}{\sigma \blacktriangleright x \mathbin{@} \eta \mapsto_{\mathsf{E}} \sigma \blacktriangleleft v}\ Var_{\mathsf{E}}$$

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

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

$$\frac{}{\sigma; \square_\eta\ e_2 \blacktriangleleft [\eta', \lambda x{:}A.\, e] \mapsto_{\mathsf{E}} \sigma; [\eta', \lambda x{:}A.\, e]\ \square \blacktriangleright e_2 \mathbin{@} \eta}\ Arg_{\mathsf{E}}$$

$$\frac{}{\sigma; [\eta, \lambda x{:}A.\, e]\ \square \blacktriangleleft v \mapsto_{\mathsf{E}} \sigma \blacktriangleright e \mathbin{@} \eta, x \hookrightarrow v}\ App_{\mathsf{E}}$$

Question 3. **[4 pts]** We want to extend the abstract machine E with an expression $\mathsf{fun}\ f\ x{:}A.\,e$, which denotes a recursive function f with a formal argument x of type A and a body e. The abstract syntax for the abstract machine E now allows $\mathsf{fun}\ f\ x{:}A.\,e$ as an expression. Extend the following definitions for the abstract machine E. Fill in the blank:

$$\begin{array}{lrcl} \mathsf{expression} & e & ::= & \cdots \mid \mathsf{fun}\ f\ x{:}A.\,e \\ \mathsf{value} & v & ::= & \cdots \mid \underline{[\eta, \mathsf{fun}\ f\ x{:}A.\,e]} \\ \mathsf{frame} & \phi & ::= & \cdots \mid \underline{[\eta, \mathsf{fun}\ f\ x{:}A.\,e]\ \square} \end{array}$$

Question 4. **[6 pts]** Give three new transition rules for $\mathsf{fun}\ f\ x{:}A.\,e$:

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

$$\frac{}{\sigma; \square_\eta\ e_2 \blacktriangleleft [\eta', \mathsf{fun}\ f\ x{:}A.\,e] \mapsto_{\mathsf{E}} \sigma; [\eta', \mathsf{fun}\ f\ x{:}A.\,e]\ \square \blacktriangleright e_2\ @\ \eta}\ \mathit{Arg}^R_{\mathsf{E}}$$

$$\frac{}{\sigma; [\eta, \mathsf{fun}\ f\ x{:}A.\,e]\ \square \blacktriangleleft v \mapsto_{\mathsf{E}} \sigma \blacktriangleright e\ @\ \eta, f \hookrightarrow [\eta, \mathsf{fun}\ f\ x{:}A.\,e], x \hookrightarrow v}\ \mathit{App}^R_{\mathsf{E}}$$

4 De Bruijn expressions [13 pts]

Question 1. [3 pts] Convert the following expression into a de Bruijn expression. Fill in the blank:

$$\lambda x.\, \lambda y.\, (\lambda z.\, (\lambda u.\, z)\ z)\ (\lambda w.\, x\ y\ w)$$

$\equiv_{\mathsf{dB}}$ $\lambda.\, \lambda.\, (\lambda.\, (\lambda.\, 1)\ 0)\ (\lambda.\, 2\ 1\ 0)$

Question 2. [10 pts] Given the following inductive definition of de Bruijn expressions, complete the definition of substitution $\sigma_n(M, N)$ and shifting $\tau_i^n(N)$ as given in the Course Notes.

de Bruijn expression	M	::=	$n \mid \lambda.\, M \mid M\ M$
de Bruijn index	n	::=	$0 \mid 1 \mid 2 \mid \cdots$

$$\sigma_n(M_1\ M_2, N) = \sigma_n(M_1, N)\ \sigma_n(M_2, N)$$

$$\sigma_n(\lambda.\, M, N) = \lambda.\, \sigma_{n+1}(M, N)$$

$$\sigma_n(m, N) = m \quad \text{if } m < n$$

$$\sigma_n(n, N) = \tau_0^n(N)$$

$$\sigma_n(m, N) = m - 1 \quad \text{if } m > n$$

$$\tau_i^n(N_1\ N_2) = \tau_i^n(N_1)\ \tau_i^n(N_2)$$

$$\tau_i^n(\lambda.\, N) = \lambda.\, \tau_{i+1}^n(N)$$

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

$$\tau_i^n(m) = m \quad \text{if } m < i$$

5 Exceptions [10 pts]

Consider the following simply typed λ-calculus extended with exceptions:

$$\begin{array}{rrcl} \textsf{type} & A & ::= & P \mid A \to A \\ \textsf{base type} & P & & \\ \textsf{expression} & e & ::= & x \mid \lambda x\!:\!A.\, e \mid e\ e \mid \textsf{try } e \textsf{ with } e \mid \textsf{exn} \\ \textsf{value} & v & ::= & \lambda x\!:\!A.\, e \end{array}$$

Question 1. **[5 pts]** Define the call-by-value operational semantics for the constructs for exceptions. You have to write only those rules related with $\textsf{try } e \textsf{ with } e'$ and exn. You will need exactly five rules.

$$\frac{}{\textsf{exn } e \mapsto \textsf{exn}}\ \mathit{Exn}$$

$$\frac{}{(\lambda x\!:\!A.\, e)\ \textsf{exn} \mapsto \textsf{exn}}\ \mathit{Exn'}$$

$$\frac{e_1 \mapsto e_1'}{\textsf{try } e_1 \textsf{ with } e_2 \mapsto \textsf{try } e_1' \textsf{ with } e_2}\ \mathit{Try}$$

$$\frac{}{\textsf{try } v \textsf{ with } e \mapsto v}\ \mathit{Try'}$$

$$\frac{}{\textsf{try exn with } e \mapsto e}\ \mathit{Try''}$$

Question 2. **[5 pts]** Assuming the call-by-value reduction strategy, give rules for propagating exceptions raised within those constructs for product types. You will need exactly four rules.

$$\begin{array}{rrcl} \textsf{type} & A & ::= & \cdots \mid A \times A \\ \textsf{expression} & e & ::= & \cdots \mid (e, e) \mid \textsf{fst } e \mid \textsf{snd } e \mid \\ \textsf{value} & v & ::= & \cdots \mid (v, v) \end{array}$$

$$\frac{}{(\textsf{exn}, e) \mapsto \textsf{exn}}$$

$$\frac{}{(v, \textsf{exn}) \mapsto \textsf{exn}}$$

$$\frac{}{\textsf{fst exn} \mapsto \textsf{exn}}$$

$$\frac{}{\textsf{snd exn} \mapsto \textsf{exn}}$$

6 Continuations [20 pts]

Consider the following simply typed λ-calculus augmented with continuations.

$$\begin{array}{rrcl}
\textsf{type} & A & ::= & P \mid A \to A \mid A \ \textsf{cont} \\
\textsf{base type} & P & & \\
\textsf{expression} & e & ::= & x \mid \lambda x{:}A.\, e \mid e\ e \mid \textsf{callcc}\ x.\, e \mid \textsf{throw}\ e\ \textsf{to}\ e \mid \langle \kappa \rangle \\
\textsf{value} & v & ::= & \lambda x{:}A.\, e \mid \langle \kappa \rangle \\
\textsf{evaluation context} & \kappa & ::= & \Box \mid \kappa\ e \mid (\lambda x{:}A.\, e)\ \kappa \mid \textsf{throw}\ \kappa\ \textsf{to}\ e \mid \textsf{throw}\ v\ \textsf{to}\ \kappa
\end{array}$$

We omit the type annotation for x in $\textsf{callcc}\ x.\, e$ because we do not need it for this problem. (In the Course Notes, we annotate x with its type as in $\textsf{callcc}\ x_{:A\ \textsf{cont}}.\, e$.)

Question 1. [5 pts] Give the reduction rules for $\textsf{callcc}\ x.\, e$ and $\textsf{throw}\ e\ \textsf{to}\ e'$:

$$\frac{}{\kappa[\![\textsf{callcc}\ x.\, e]\!] \mapsto \kappa[\![[\langle \kappa \rangle / x]e]\!]}\ \textit{Callcc}$$

$$\frac{}{\kappa[\![\textsf{throw}\ v\ \textsf{to}\ \langle \kappa' \rangle]\!] \mapsto \kappa'[\![v]\!]}\ \textit{Throw}$$

Question 2. [10 pts] Assuming the call-by-value strategy, give the result of evaluating each expression below. We assume integer constants and the addtion operator + defined as usual.

$\textsf{fst callcc}\ x.\, (\textsf{true}, \textsf{throw}\ (\textsf{false}, \textsf{false})\ \textsf{to}\ x) \quad \mapsto^* \quad$ false

$\textsf{snd callcc}\ x.\, (\textsf{throw}\ (\textsf{true}, \textsf{true})\ \textsf{to}\ x, \textsf{false}) \quad \mapsto^* \quad$ true

$1\ +\ \textsf{callcc}\ x.\, (2\ +\ (\textsf{throw}\ 3\ \textsf{to}\ x)) \quad \mapsto^* \quad$ 4

$1\ +\ \textsf{callcc}\ x.\, \textsf{if}\ (\textsf{throw}\ 2\ \textsf{to}\ x)\ \textsf{then}\ 3\ \textsf{else}\ 4 \quad \mapsto^* \quad$ 3

Question 3. [5 pts] Complete the typing rules for $\textsf{callcc}\ x.\, e$ and $\textsf{throw}\ e_1\ \textsf{to}\ e_2$:

$$\frac{\Gamma, x : A\ \textsf{cont} \vdash e : A}{\Gamma \vdash \textsf{callcc}\ x.\, e : A}\ \textsf{Callcc}$$

$$\frac{\Gamma \vdash e_1 : A \quad \Gamma \vdash e_2 : A\ \textsf{cont}}{\Gamma \vdash \textsf{throw}\ e_1\ \textsf{to}\ e_2 : C}\ \textsf{Throw}$$

7 Subtyping [10 pts]

Question 1. **[5 pts]** The principle of subtyping is a principle specifying when a type is a subtype of another type. It states that A is a subtype of B if an expression of type A may be used wherever an expression of type B is expected. Formally we write $A \leq B$ if A is a subtype of B. The principle of subtyping justifies two subtyping rules: reflexivity and transitivity. Write the two subtyping rules:

$$\frac{}{A \leq A}\ \mathit{Refl}_{\leq}$$

$$\frac{A \leq B \quad B \leq C}{A \leq C}\ \mathit{Trans}_{\leq}$$

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

$$\frac{\Gamma \vdash e : A \qquad A \leq B}{\Gamma \vdash e : B}\ \mathsf{Sub}$$

Question 2. **[5 pts]** Complete subtyping rules for product, function and reference types.

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

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

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

8 Recursive Types [25 pts]

Consider the following simply typed λ-calculus extended with recursive types.

$$\begin{array}{rrcl} \textsf{type} & A & ::= & \textsf{unit} \mid A \rightarrow A \mid A + A \mid \alpha \mid \mu\alpha.A \\ \textsf{expression} & e & ::= & x \mid \lambda x{:}A.\, e \mid e\ e \mid () \mid \textsf{inl}_A\ e \mid \textsf{inr}_A\ e \mid \\ & & & \textsf{case}\ e\ \textsf{of inl}\ x.\, e \mid \textsf{inr}\ x.\, e \mid \textsf{fold}_C\ e \mid \textsf{unfold}_C\ e \\ \textsf{typing context} & \Gamma & ::= & \cdot \mid \Gamma, x : A \mid \Gamma, \alpha\ \textsf{type} \end{array}$$

Question 1. **[5 pts]** Given a recursive type $C = \mu\alpha.A$, $\textsf{fold}_C\ e$ and $\textsf{unfold}_C\ e$ convert $[C/\alpha]A$ to C and vice versa, respectively. Complete typing rules for $\textsf{fold}_C\ e$ and $\textsf{unfold}_C\ e$:

$$\frac{C = \mu\alpha.A \qquad \Gamma \vdash e : [C/\alpha]A \qquad \Gamma \vdash C\ \textsf{type}}{\Gamma \vdash \textsf{fold}_C\ e : C}\ \textsf{Fold}$$

$$\frac{C = \mu\alpha.A \qquad \Gamma \vdash e : C}{\Gamma \vdash \textsf{unfold}_C\ e : [C/\alpha]A}\ \textsf{Unfold}$$

Question 2. **[5 pts]** Define the call-by-value operational semantics of those constructs for recursive types. You have to write only those rules related with $\textsf{fold}_C\ e$ and $\textsf{unfold}_C\ e$. You will need exactly three rules.

$$\frac{e \mapsto e'}{\textsf{fold}_C\ e \mapsto \textsf{fold}_C\ e'}\ \textit{Fold}$$

$$\frac{e \mapsto e'}{\textsf{unfold}_C\ e \mapsto \textsf{unfold}_C\ e'}\ \textit{Unfold}$$

$$\frac{}{\textsf{unfold}_C\ \textsf{fold}_C\ v \mapsto v}\ \textit{Unfold}^2$$

Question 3. [7 pts] Translate a recursive datatype for natural numbers in the above simply typed λ-calculus.

$$\texttt{datatype nat} \;=\; \texttt{Zero} \mid \texttt{Succ of nat}$$

$$\texttt{nat} \;=\; \underline{\mu\alpha.\mathsf{unit}{+}\alpha}$$

$$\texttt{Zero} \;=\; \underline{\mathsf{fold}_{\texttt{nat}}\ \mathsf{inl}_{\texttt{nat}}\ ()}$$

$$\texttt{Succ}\ e \;=\; \underline{\mathsf{fold}_{\texttt{nat}}\ \mathsf{inr}_{\mathsf{unit}}\ e}$$

$$\texttt{case}\ e\ \texttt{of Zero} \Rightarrow e_1 \mid \texttt{Succ}\ x \Rightarrow e_2 \;=\; \underline{\mathsf{case}\ \mathsf{unfold}_{\texttt{nat}}\ e\ \mathsf{of}\ \mathsf{inl}\ \_.\, e_1 \mid \mathsf{inr}\ x.\, e_2}$$

Question 4. [8 pts] Translate the untyped λ-calculus to the simply typed λ-calculus with the following definition:

$$\begin{array}{rrcl} \mathsf{type} & A & ::= & A{\to}A \mid \alpha \mid \mu\alpha.A \\ \mathsf{expression} & e & ::= & x \mid \lambda x{:}A.\, e \mid e\ e \mid \mathsf{fold}_A\ e \mid \mathsf{unfold}_A\ e \end{array}$$

We translate an expression e in the untyped λ-calculus to an expression e° in the simply typed λ-calculus. We treat all expressions in the untyped λ-calculus alike by assigning a unique type Ω (*i.e.*, e° is to have type Ω). If every expression is assigned type Ω, we may think that $\lambda x.\, e$ is assigned type $\Omega{\to}\Omega$ *as well as* type Ω. Or, in order for $e_1\ e_2$ to be assigned type Ω, e_1 must be assigned *not only* type Ω but also type $\Omega{\to}\Omega$ because e_2 is assigned type Ω. Thus Ω must be identified with $\Omega{\to}\Omega$.

Use recursive types and their constructs to complete the definition of Ω and e°. Fill in the blank:

$$\Omega \;=\; \underline{\mu\alpha.\alpha{\to}\alpha}$$

$$x^\circ \;=\; \underline{x}$$

$$(\lambda x.\, e)^\circ \;=\; \underline{\mathsf{fold}_\Omega\ \lambda x{:}\Omega.\, e^\circ}$$

$$(e_1\ e_2)^\circ \;=\; \underline{(\mathsf{unfold}_\Omega\ {e_1}^\circ)\ {e_2}^\circ}$$

9 Polymorphism [15 pts]

Question 1. [**2 pts**] Consider the following SML example. The expression at the prompt leads to a warning message. Explain with no more than two sentences why this warning message is printed. You may write in Korean.

```
- let val id = (fn y => y) (fn z => z) in id end;
stdIn:1.1-1.46 Warning: type vars not generalized because of
   value restriction are instantiated to dummy types (X1,X2,...)
val it = fn : ?.X1 -> ?.X1
```

Answer: Due to the value restriction, `id` must have a monotype. But SML system cannot determine a unique monotype for `id`.

Question 2. [**3 pts**] Consider the following SML example. The expression at the prompt is ill-typed and raises an error. Explain with no more two sentences why this error occurs. You may write in Korean.

```
- let val id = (fn y => y) (fn z => z) in (id true, id 1) end;
stdIn:2.40-2.56 Error: operator and operand don't agree [literal]
  operator domain: bool
  operand:         int
  in expression:
    id 1
```

Answer: Due to the value restriction, `id` cannot have a polytype such as $\forall\alpha.\alpha\to\alpha$. Therefore `id` cannot be applied to two values `true` and `1` of different types at the same time.

Question 3. [**10 pts**] Consider System F and the predicate polymorphic λ-calculus given in the Course Notes. For each untyped expression given below,

- write F if it is typable only in System F,
- write P if it is typable only in the predicate λ-calculus,
- write FP if it is typable in both systems, or
- write N if it is typable in neither system.

$\lambda x.\, x$	FP
$\lambda x.\, x\ x$	F
$(\lambda x.\, x\ x)\ (\lambda x.\, x\ x)$	N
$(\lambda f.\, (f\ \mathsf{true}, f\ 0))\ (\lambda x.\, x)$	F

10 Type Reconstruction [20 pts]

Consider the implicit let-polymorphic type system given in the Course Notes.

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

- $S \cdot U$ 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.
- An auxiliary function $\mathsf{Gen}_\Gamma(A)$ generalizes monotype A to a polytype after taking into account free type variables in typing context Γ:

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

- 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.

Question 1. [3 pts] Fill in the blank:

$$\begin{array}{rcl}
\mathsf{Gen}.(\alpha \to \alpha) & = & \forall \alpha.\alpha \to \alpha \\
\mathsf{Gen}_{x:\alpha}(\alpha \to \alpha) & = & \underline{\qquad \alpha \to \alpha \qquad} \\
\mathsf{Gen}_{x:\alpha}(\alpha \to \beta) & = & \underline{\qquad \forall \beta.\alpha \to \beta \qquad} \\
\mathsf{Gen}_{x:\alpha,y:\beta}(\alpha \to \beta) & = & \underline{\qquad \alpha \to \beta \qquad}
\end{array}$$

Question 2. [3 pts] Complete the unification algorithm Unify:

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

Question 3. **[4 pts]** Now we add a base type int, and thus have $A \ ::= \ A \to A \mid \alpha \mid \mathsf{int}$. Then what is the additional case for Unify?

$$\mathsf{Unify}(E, \mathsf{int} = A) = \mathsf{Unify}(E, A = \mathsf{int}) \quad = \quad \text{if } A = \mathsf{int} \text{ then } \mathsf{Unify}(E) \text{ else } \mathit{fail}$$

Question 4. **[4 pts]** Complete the algorithm $\mathcal{W}$:

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

Question 5. **[6 pts]** Now we add an untyped fixed point construct $\mathsf{fix}\ x.\, e$. The typing rule for $\mathsf{fix}\ x.\, e$ is as follows:

$$\frac{\Gamma, x : A \triangleright e : A}{\Gamma \triangleright \mathsf{fix}\ x.\, e : A}\ \mathsf{Fix}$$

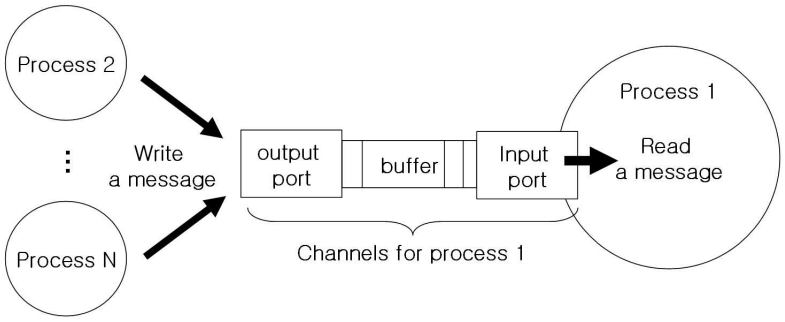
Complete the case for $\mathsf{fix}\ x.\, e$ in the algorithm $\mathcal{W}$:

$$\begin{aligned}
\mathcal{W}(\Gamma, \mathsf{fix}\ x.\, e) &= \text{let } (S_1, A_1) = \underline{\mathcal{W}(\Gamma + x : \alpha, e)} \text{ in} && (\text{fresh } \alpha) \\
&\quad \text{let } S_2 = \underline{\mathsf{Unify}(S_1 \cdot \alpha = A_1)} \text{ in} \\
&\quad (\underline{S_2 \circ S_1}, \underline{S_2 \cdot A_1})
\end{aligned}$$

11 Subtyping on channels [15 pts]

In parallel computations, two or more processes that run in parallel may communicate with each other. For example, one process may send a message, such as data or code, to another process. One way to support communications is to use channels.

A channel is a buffer with input and output ports which messages can be read from and written to. Every channel belongs to a certain process. Only the process that owns the channel may read messages from the input port of the channel, while every process may write messages to the output port of the channel. The following figure illustrates how channel reads and writes work:



In order to exploit channels, we extend the simply typed λ-calculus with constructs and types for channels. For this problem, we do not consider constructs for channels since our goal is to devise subtyping rules for channels. As in the Course Notes, we write $A \leq B$ if A is a subtype of B. Let us assume two base types nat and int, and a subtyping relation $\mathsf{nat} \leq \mathsf{int}$. For this problem, we consider only base types and channel types:

$$\begin{array}{rrcl} \textsf{type} & A, B & ::= & P \mid \cdots \mid \langle A, B \rangle \\ \textsf{base type} & P & ::= & \mathsf{nat} \mid \mathsf{int} \end{array}$$

Channel types take the form $\langle A, B \rangle$ which is a pair consisting of an input port type A and an output port type B. For channels of type $\langle A, B \rangle$, only values of type A can be read from it while only values of type B can be written to it.

It turns out that not every channel type is valid. Suppose a channel of type $\langle \mathsf{nat}, \mathsf{int} \rangle$. Then only values of type int can be written to the output port of it, while only values of type nat can be read from the input port. If `-3` of type int is written to the output port, however, a type error will be raised because only values of type nat can be read from the input port while `-3` is not of type nat. Therefore we introduce a new form of declaration to ensure the validity of channel types: a type declaration A type meaning that A is a valid type. Every base type P is a valid type.

Question 1. **[15 pts]** Complete two rules for ensuring the validity of channel types and subtyping principle. We assume that A and B are valid types, and $\langle A_1, B_1 \rangle$ and $\langle A_2, B_2 \rangle$ are valid channel types.

$$\frac{B \leq A}{\langle A, B \rangle \ \mathsf{type}} \qquad \frac{A_1 \leq A_2 \quad B_2 \leq B_1}{\langle A_1, B_1 \rangle \leq \langle A_2, B_2 \rangle}$$

12 A pathfinding machine [22 pts]

In this problem you will design a pathfinding machine, similar to the abstract machine C, which takes a binary tree whose leaf nodes can be designated as exits and searches for all paths from the root node to exits. Here is an example of such a binary tree:

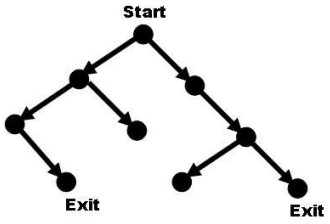


Figure 1: An example tree

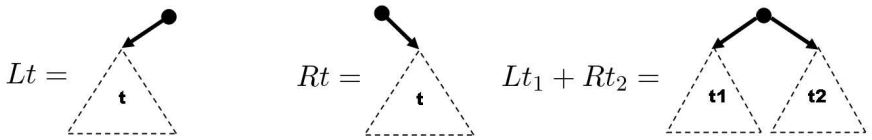
A binary tree for this problem should satisfy the following rule:

- A non-leaf node cannot be an exit.

Here is the abstract syntax for binary trees:

$$\text{tree} \quad t \ ::= \ [\cdot] \mid [exit] \mid Lt \mid Rt \mid (Lt + Rt)$$

- $[\cdot]$ represents a leaf node that is not an exit.
- $[exit]$ represents a leaf node that is an exit.
- Lt represents a tree whose root node has t as its left subtree but has no right subtree.
- Rt represents a tree whose root node has t as its right subtree but has no left subtree.
- $Lt_1 + Rt_2$ represents a tree whose root node has t_1 as its left subtree and t_2 as its right subtree.



For example, the tree T in Figure 1 is represented as follows:

$$T = (L(LR[exit] + R[\cdot]) + RR(L[\cdot] + R[exit]))$$

Question 1. [10 pts] We want to implement a function $Find(t)$ which finds the set of all paths from the root node to exits. In order to represent a path, we use the following definition of direction and lpath:

$$\begin{array}{rrcl} \text{direction} & d & ::= & L \mid R \\ \text{lpath} & \pi & ::= & \epsilon \mid d\pi \end{array}$$

- direction means a direction of traversal. L means "go to the left subtree" and R "go to the right subtree."

- lpath represents a sequence of directions which is the history of traversal. ϵ denotes an empty sequence (*i.e*, $\epsilon\pi = \pi = \pi\epsilon$).

For example, lpath generates such paths as:

$$\epsilon,\ L,\ R,\ LLR,\ RRR,\ RLRR, ...$$

For the tree T in Figure 1, paths that lead to the two exits are represented as LLR and RRR. Thus we have $Find(T) = \{LLR,\ RRR\}$. Fill in the blank to complete the definition of the function $Find(t)$. Use the conventional set notation:

$$\begin{aligned}
Find([\cdot]) &= \underline{\qquad\qquad \{\} \qquad\qquad} \\
Find([exit]) &= \underline{\qquad\qquad \{\epsilon\} \qquad\qquad} \\
Find(Lt) &= \underline{\{L\pi \mid \pi \in Find(t)\}} \\
Find(Rt) &= \underline{\{R\pi \mid \pi \in Find(t)\}} \\
Find(Lt_1 + Rt_2) &= \underline{\{L\pi \mid \pi \in Find(t_1)\} \cup \{R\pi \mid \pi \in Find(t_2)\}}
\end{aligned}$$

Question 2. [12 pts] We want to design a pathfinding machine that computes a path from the root node point to a certain exit in a given binary tree. We use the following definition for this machine:

$$\begin{array}{rrcl}
\textsf{path} & \phi & ::= & \epsilon \mid \phi d \\
\textsf{frame} & \psi & ::= & \diamond \mid t \\
\textsf{stack} & \sigma & ::= & \square \mid \sigma; \psi \\
\textsf{state} & s & ::= & \sigma \blacktriangleright (\phi,\ t) \mid \sigma \blacktriangleleft \phi \mid \ominus(\phi)
\end{array}$$

- path is equivalent to lpath. Note the position of d in ϕd.
- frame is an element of the stack and denotes an unexplored subtree. $\diamond$ means that there is no unexplored subtree. $\diamond$ is generated when the machine encounters a tree with only one subtree (*e.g.* Lt or Rt). If the machine encounters a tree with both subtrees, the machine chooses one direction and generate a frame t to memorize the subtree t in the other direction.
- stack represents a sequence of frames and keeps a sequence of unexplored subtrees which are needed when the machine backtracks. When the machine is currently backtracking and the rightmost frame of stack is $\diamond$, the machine continues to backtrack because it means that there is no subtree to explore yet. If the machine is currently backtracking and the rightmost frame of stack is t, the machine will start to explore tree t.
- state represents a state of the machine. '$\sigma \blacktriangleright (\phi,\ t)$' means that the machine is currently exploring tree t. σ represents the sequence of unexplored subtrees, and ϕ represents the sequence of directions that has been accumulated until the current state is reached. In this state, the machine will analyze tree t and determine what to do next. '$\sigma \blacktriangleleft \phi$' means that the machine is currently backtracking. By analyzing the stack σ, the machine will determine what to do next. '$\ominus(\phi)$' means that the machine has found an exit. ϕ is the path from the root node to the exit, which is the final outcome.

The goal of this problem is to write the rules for the state transition judgment $s \mapsto s'$ for this machine. We give some specifications for the machine below:

- This machine finds only one path that leads to an exit. That is, as soon as the machine finds the first exit, it terminates.
- The machine always tries to explore the *left subtree first*. If there is no left subtree, it tries to explore the right subtree. If there is no subtree to explore, the machine starts to backtrack.
- For a given tree t, the initial machine state is $\square \blacktriangleright (\epsilon, t)$
- If there is no exit in a given tree, the final machine state is $\square \blacktriangleleft \epsilon$
- If there exists at least one exit in a given tree, the final machine state contains the path that leads to the first exit. For example, for the tree T in Figure 1, the final machine state is $\ominus(LLR)$.

We give the *Final* rule below. Fill in the blank and complete the remaining six rules for this machine:

$$\frac{}{\sigma \blacktriangleright (\phi, [exit]) \mapsto \ominus(\phi)}\ Final$$

$$\frac{}{\sigma \blacktriangleright (\phi, Lt) \mapsto \sigma; \diamond \blacktriangleright (\phi L, t)}\ Move_{\mathsf{L}}$$

$$\frac{}{\sigma \blacktriangleright (\phi, Rt) \mapsto \sigma; \diamond \blacktriangleright (\phi R, t)}\ Move_{\mathsf{R}}$$

$$\frac{}{\sigma \blacktriangleright (\phi, (Lt_1 + Rt_2)) \mapsto \sigma; Rt_2 \blacktriangleright (\phi L, t_1)}\ Move_{\mathsf{LR}}$$

$$\frac{}{\sigma \blacktriangleright (\phi, [\cdot]) \mapsto \sigma \blacktriangleleft \phi}\ Move_{\mathsf{empty}}$$

$$\frac{}{\sigma; \diamond \blacktriangleleft \phi d \mapsto \sigma \blacktriangleleft \phi}\ Back_{\mathsf{dot}}$$

$$\frac{}{\sigma; Rt \blacktriangleleft \phi d \mapsto \sigma; \diamond \blacktriangleright (\phi R, t)}\ Back_{\mathsf{R}}$$

Work sheet