Name: Hemos ID:

# CSE-321 Programming Languages 2010
# Final

| | Prob 1 | Prob 2 | Prob 3 | Prob 4 | Prob 5 | Prob 6 | Total |
|---|---|---|---|---|---|---|---|
| Score | | | | | | | |
| Max | 18 | 28 | 16 | 12 | 36 | 40 | 150 |

- There are six problems on 16 pages, including two work sheets, in this exam.
- The maximum score for this exam is 150 points, and there is an extracredit problem.
- 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 2010`$^\lambda$ [Extracredit]

State "Yes" if you wear the `PL 2010`$^\lambda$ T-shirt. ______________________

## PL 2010 Tekken Match [Extracredit]

State "Yes" if you played in PL 2010 Tekken Match. ______________________

Who did you beat in PL 2010 Tekken Match? ______________________

# 1 Mutable references [18 pts]

Consider the following simply-typed $\lambda$-calculus extended with mutable references.

$$\begin{array}{rrcl}
\textsf{type} & A & ::= & P \mid A \rightarrow A \mid \textsf{int} \mid \textsf{ref}\ A \\
\textsf{expression} & e & ::= & x \mid \lambda x\!:\!A.\, e \mid e\ e \mid \textsf{let}\ x = e\ \textsf{in}\ e \mid \textsf{ref}\ e \mid !e \mid e := e \mid 0 \mid 1 \mid \cdots \\
\textsf{value} & v & ::= & \lambda x\!:\!A.\, e \mid l \mid 0 \mid 1 \mid \cdots \\
\textsf{store} & \psi & ::= & \cdot \mid \psi, l \mapsto v \\
\textsf{typing context} & \Gamma & :;= & \cdot \mid \Gamma, x : A \\
\textsf{store typing context} & \Psi & ::= & \cdot \mid \Psi, l \mapsto A
\end{array}$$

**Question 1.** **[8 pts]** We want to represent an array of integers as a function taking an index (of type int) and returning a corresponding elements of the array. We choose a functional representation of arrays by defining type iarray for arrays of integers as follows:

$$\textsf{iarray} = \textsf{ref}\ (\textsf{int} \rightarrow \textsf{int})$$

We need the following constructs for arrays:

- new : unit $\rightarrow$ iarray for creating a new array.
  new () returns a new array of indefinite size; all elements are initialized as 0.
- access : iarray $\rightarrow$ int $\rightarrow$ int for accessing an array.
  access $a$ $i$ returns the $i$-th element of array $a$.
- update : iarray $\rightarrow$ int $\rightarrow$ int $\rightarrow$ 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:

$$\begin{array}{rcl}
\textsf{new} & = & \lambda _\!:\!\textsf{unit}.\ \textsf{ref}\ \lambda i\!:\!\textsf{int}.\, 0 \\[1em]
\textsf{access} & = & \lambda a\!:\!\textsf{iarray}.\, \lambda i\!:\!\textsf{int}.\, (!a)\ i \\[1em]
\textsf{update} & = & \lambda a\!:\!\textsf{iarray}.\, \lambda i\!:\!\textsf{int}.\, \lambda n\!:\!\textsf{int}.
\end{array}$$

________________________________________

________________________________________

**Question 2.** **[10 pts]** State progress and type preservation theorems. In your statements, 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 $e$ with store $\psi$ reduces to $e'$ with $\psi'$.
- A store judgment $\psi :: \Psi$ means that $\Psi$ corresponds to $\psi$.

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

*(1)* __________________*, or*

*(2) for any* ______________________ *such that* ______________________*,*

*there exist some* ______________________ *and* __________________ *such that* ______________________*.*

**Theorem (Type preservation).** *Suppose* $\left\{ \begin{array}{l} \Gamma \mid \Psi \vdash e : A \\ \psi :: \Psi \\ e \mid \psi \mapsto e' \mid \psi' \end{array} \right.$.

*Then there exists a store typing context $\Psi'$ such that* $\left\{ \begin{array}{l} \rule{6cm}{0.4pt} \\ \\ \rule{5cm}{0.4pt} \\ \\ \rule{5cm}{0.4pt} \end{array} \right.$.

## 2 Evaluation context and environment [28 pts]

Consider the following fragment of the simply-typed $\lambda$-calculus.

$$
\begin{array}{rrcl}
\textsf{type} & A & ::= & P \mid A \rightarrow A \\
\textsf{base type} & P & ::= & \textsf{bool} \\
\textsf{expression} & e & ::= & x \mid \lambda x\!:\!A.\, e \mid e\ e \mid \textsf{true} \mid \textsf{false} \mid \textsf{if}\ e\ \textsf{then}\ e\ \textsf{else}\ e
\end{array}
$$

**Question 1. [5 pts]** Give the definition of evaluation contexts for the call-by-value strategy.

evaluation context $\kappa$ ::= ________________________________________

**Question 2. [5 pts]** Give the definition of evaluation contexts for the call-by-name strategy.

evaluation context $\kappa$ ::= ________________________________________

**Question 3. [5 pts]** Under the call-by-value strategy, give an expression $e$ such that

- $e = \kappa[\![e']\!]$ where $e'$ is the redex, and
- $e$ reduces to $e_0$ that is decomposed to $\kappa[\![e'']\!]$ where $e''$ is the redex for the next reduction.

________________________________________

**Question 4. [5 pts]** Under the call-by-value strategy, give an expression $e$ such that

- $e = \kappa[\![e']\!]$ where $e'$ is the redex, and
- $e$ reduces to $e_0$ that is decomposed to $\kappa'[\![e'']\!]$ where $e''$ is the redex for the next reduction and $\kappa \neq \kappa'$.

________________________________________

**Question 5. [8 pts]** The key idea behind the environment semantics is to postpone a substitution $[v/x]e$ by storing a pair of value $v$ and variable $x$ in an *environment*. We use the following definition of environment:

$$\textsf{environment} \qquad \eta \quad ::= \quad \cdot \mid \eta, x \hookrightarrow v$$

$\cdot$ denotes an empty environment, and $x \hookrightarrow v$ means that variable $x$ is to be replaced by value $v$. We use an *environment evaluation judgment* of the form $\eta \vdash e \hookrightarrow v$:

$$\eta \vdash e \hookrightarrow v \qquad \Leftrightarrow \qquad e \textit{ evaluates to } v \textit{ under environment } \eta$$

Give the definition of values for the simply-typed $\lambda$-calculus given in the beginning of this section.

$$\textsf{value} \qquad v \quad ::= \quad \underline{\hspace{20em}}$$

Complete the following three rules for the environment evaluation judgment $\eta \vdash e \hookrightarrow v$ corresponding to the call-by-value strategy.

$$\frac{\hspace{18em}}{\eta \vdash x \hookrightarrow \hspace{6em}}$$

$$\frac{\hspace{18em}}{\eta \vdash \lambda x\!:\!A.\, e \hookrightarrow \hspace{6em}}$$

$$\frac{\hspace{28em}}{\eta \vdash e_1\ e_2 \hookrightarrow \hspace{6em}}$$

## 3 Subtyping [16 pts]

**Question 1. [6 pts]** Complete subtyping rules for function and reference types.

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

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

**Question 2. [10 pts]** The Java language adopts the following subtyping rule for array types:

$$\frac{A \leq B}{\mathsf{array}\ A \leq \mathsf{array}\ B}\ \mathit{Array}_{\leq}'$$

While it is controversial whether the rule $\mathit{Array}_{\leq}'$ is a flaw in the design of the Java language, using the rule $\mathit{Array}_{\leq}'$ for subtyping on array types incurs a runtime overhead which would otherwise be unnecessary. State specifically when and why such runtime overhead occurs in terms of dynamic tag-checks which inspect type information of each object at runtime. You may write in Korean.

# 4 Recursive types [12 pts]

Consider the following simply-typed $\lambda$-calculus extended with recursive types:

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

**Question 1.** **[6 pts]** Give typing rules for $\textsf{fold}_C\ e$ and $\textsf{unfold}_C\ e$.

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

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

**Question 2.** **[6 pts]** Consider the following recursive datatype for natural numbers:

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

Using a recursive type, we encode type `nat` as $\mu\alpha.\textsf{unit}+\alpha$. Encode `Zero` and `Succ` $e$.

`Zero` $=$ ________________________________________

`Succ` $e$ $=$ ________________________________________

**Question 3. [Extracredit]** We want to translate an expression $e$ in the untyped $\lambda$-calculus into an expression $e^\circ$ in the simply typed $\lambda$-calculus extended with recursive types. We treat all expressions in the untyped $\lambda$-calculus alike by assigning a unique type $\Omega$ (*i.e.*, $e^\circ$ is to have type $\Omega$). If every expression is assigned type $\Omega$, we may think that $\lambda x.\, e$ is assigned type $\Omega \rightarrow \Omega$ *as well as* type $\Omega$. Or, in order for $e_1\ e_2$ to be assigned type $\Omega$, $e_1$ must be assigned *not only* type $\Omega$ but also type $\Omega \rightarrow \Omega$ because $e_2$ is assigned type $\Omega$. Thus $\Omega$ must be identified with $\Omega \rightarrow \Omega$.

Use recursive types and their constructs to complete the definition of $\Omega$ and $e^\circ$. Fill in the blank:

$$
\begin{array}{rcl}
\Omega & = & \underline{\hspace{8cm}} \\
x^\circ & = & x \\
(\lambda x.\, e)^\circ & = & \underline{\hspace{8cm}} \\
(e_1\ e_2)^\circ & = & \underline{\hspace{8cm}}
\end{array}
$$

## 5 Polymorphism (36 pts)

The following shows the abstract syntax for System F:

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

Below we define an *erasure* function $erase(\cdot)$ which takes an expression in System F and erases all type annotations in it to produce a corresponding expression in untyped $\lambda$-calculus:

$$\begin{array}{lcl} erase(x) & = & x \\ erase(\lambda x : A.\, e) & = & \lambda x.\, erase(e) \\ erase(e_1\ e_2) & = & erase(e_1)\ erase(e_2) \\ erase(\Lambda \alpha.\, e) & = & erase(e) \\ erase(e\, [\![A]\!]) & = & erase(e) \end{array}$$

**Question 1.** **[5 pts]** Give a well-typed closed expression $e$ in System F such that $erase(e) = \lambda x.\, x\ x$. If there is no such expression, state so.

______________________________

**Question 2.** **[5 pts]** Give a well-typed closed expression $e$ in System F such that $erase(e) = (\lambda x.\, x\ x)\ (\lambda x.\, x\ x)$. If there is no such expression, state so.

______________________________

**Question 3.** **[6 pts]** A Church numeral $\widehat{n}$ takes a function $f$ and returns another function $f^n$ which applies $f$ exactly $n$ times. In order for $f^n$ to be well-typed, its argument type and return type must be identical. Hence we define the base type $\textsf{nat}$ in System F as follows:

$$\textsf{nat} \quad = \quad \forall \alpha . (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)$$

Encode a zero $\textsf{zero}$ of type $\textsf{nat}$ and a successor function $\textsf{succ}$ of type $\textsf{nat} \rightarrow \textsf{nat}$:

$\textsf{zero}$ = ______________________________

$\textsf{succ}$ = ______________________________

The following shows the abstract syntax for the let-polymorphism system:

$$
\begin{array}{lrcl}
\textsf{monotype} & A & ::= & A \rightarrow A \mid \alpha \\
\textsf{polytype} & U & ::= & A \mid \forall \alpha . U \\
\textsf{expression} & e & ::= & x \mid \lambda x \mathbin{:} A .\, e \mid e\ e \mid \Lambda \alpha .\, e \mid e\, [\![ A ]\!] \mid \mathsf{let}\ x \mathbin{:} U = e\ \mathsf{in}\ e
\end{array}
$$

Below we define an erasure function $erase(\cdot)$ which takes an expression in the let-polymorphism system and erases all type annotations in it to produce a corresponding expression in the implicit let-polymorphism system:

$$
\begin{array}{lcl}
erase(x) & = & x \\
erase(\lambda x \mathbin{:} A .\, e) & = & \lambda x .\, erase(e) \\
erase(e_1\ e_2) & = & erase(e_1)\ erase(e_2) \\
erase(\Lambda \alpha .\, e) & = & erase(e) \\
erase(e\, [\![ A ]\!]) & = & erase(e) \\
erase(\mathsf{let}\ x \mathbin{:} U = e\ \mathsf{in}\ e') & = & \mathsf{let}\ x = erase(e)\ \mathsf{in}\ erase(e')
\end{array}
$$

**Question 4.** **[5 pts]** Give a well-typed closed expression $e$ in the let-polymorphism system such that $erase(e) = \mathsf{let}\ f = \lambda x .\, x\ \mathsf{in}\ (f\ \mathsf{true}, f\ 0)$. Assume two monotypes $\mathsf{bool}$ for boolean values and $\mathsf{int}$ for integers.

**Question 5.** **[10 pts]** Explain value restriction. You may write in Korean.

**Question 6.** **[5 pts]** Give a well-typed closed expression $e$ in the let-polymorphism system with value restriction such that $erase(e) = \mathsf{let}\ f = (\lambda x .\, x)\ (\lambda y .\, y)\ \mathsf{in}\ (f\ \mathsf{true}, f\ 1)$. If there is no such expression, explain why. You may write in Korean.

## 6 Type reconstruction [40 pts]

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

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

- $S \cdot U$ and $S \cdot \Gamma$ denote applications of $S$ to $U$ and $\Gamma$, respectively.
- $ftv(\Gamma)$ denotes the set of free type variables in $\Gamma$; $ftv(U)$ 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 $x$.

**Question 1. [6 pts]** The type reconstruction algorithm, called $\mathcal{W}$, 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. State the soundness theorem of the algorithm $\mathcal{W}$. Use the typing judgment of the form $\Gamma \triangleright x : U$ in your statement.

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

then ____________________________________________.

**Question 2.** **[14 pts]** Assume that you are given the unification algorithm $\mathsf{Unify}(E)$ and an auxiliary function $\mathsf{Gen}_\Gamma(A)$ which generalizes monotype $A$ to a polytype after taking into account free type variables in typing context $\Gamma$. The following specifies the algorithm $\mathcal{W}$. Complete the specification:

$$
\begin{array}{rcll}
\mathcal{W}(\Gamma, x) & = & (\mathrm{id}, \{\vec{\beta}/\vec{\alpha}\} \cdot A) & x : \forall\vec{\alpha}.A \in \Gamma \text{ and fresh } \vec{\beta} \\
\mathcal{W}(\Gamma, \lambda x.\, e) & = & \text{let } (S, A) = \mathcal{W}(\Gamma + x : \alpha, e) \text{ in} & \text{fresh } \alpha \\
 & & \underline{\hspace{8cm}} & \\
\mathcal{W}(\Gamma, e_1\ e_2) & = & \text{let } (S_1, A_1) = \mathcal{W}(\Gamma, e_1) \text{ in} & \\
 & & \text{let } \underline{\hspace{8cm}} & \\
 & & \text{let } \underline{\hspace{8cm}} & \\
 & & \underline{\hspace{8cm}} & \\
\mathcal{W}(\Gamma, \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2) & = & \text{let } \underline{\hspace{8cm}} & \\
 & & \text{let } \underline{\hspace{8cm}} & \\
 & & \underline{\hspace{8cm}} &
\end{array}
$$

**Question 3.** **[6 pts]** Given an application $e_1\ e_2$, the algorithm $\mathcal{W}$ reconstructs first the type of $e_1$ and then the type of $e_2$. Modify the algorithm $\mathcal{W}$ so that it reconstructs first the type of $e_2$ and then the type of $e_1$.

$$\mathcal{W}(\Gamma, e_1\ e_2) \quad = \quad \text{let } (S_2, A_2) = \mathcal{W}(\Gamma, e_2) \text{ in}$$

let ________________________________________

let ________________________________________

________________________________________

**Question 4.** **[6 pts]** Now we add a product type $A_1 \times A_2$ and an untyped pair construct $(e_1, e_2)$. The typing rule for $(e_1, e_2)$ is as follows:

$$\frac{\Gamma \triangleright e_1 : A_1 \quad \Gamma \triangleright e_2 : A_2}{\Gamma \triangleright (e_1, e_2) : A_1 \times A_2} \times\mathsf{I}$$

Complete the case for $(e_1, e_2)$ in the algorithm $\mathcal{W}$:

$$\mathcal{W}(\Gamma, (e_1, e_2)) \quad = \quad \rule{8cm}{0.4pt}$$

________________________________________

________________________________________

**Question 5.** **[8 pts]** What is the result of $\mathcal{W}(\cdot, \textsf{let } f = \lambda x.\, x \textsf{ in } (f\ 0, f\ \textsf{true}))$? Assume two monotypes $\textsf{bool}$ for boolean values and $\textsf{int}$ for integers.

Substitution = ________________________________________

Type = ________________________________________

For each type variable in the resultant type substitution, indicate where it is produced.

## Work sheet

## Work sheet