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[^] [Extracredit]**

State “Yes” if you wear the PL 2010[^] 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 λ-calculus extended with mutable references.

\[
\text{type } \quad A := P | A \rightarrow A | \text{int} | \text{ref } A \\
\text{expression } \quad e := x | \lambda x : A. e | e e | \text{let } x = e \text{ in } e | \text{ref } e | !e | e := e | 0 | 1 | \cdots \\
\text{value } \quad v := \lambda x : A. e | l | 0 | 1 | \cdots \\
\text{store } \quad \psi ::= \cdot | \psi, l \mapsto v \\
\text{typing context } \quad \Gamma ::= \cdot | \Gamma, x : A \\
\text{store typing context } \quad \Psi ::= \cdot | \Psi, l \mapsto A
\]

**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 \text{iarray} for arrays of integers as follows:

\[
iarray = \text{ref } (\text{int} \rightarrow \text{int})
\]

We need the following constructs for arrays:

- **new**: \text{unit} \rightarrow \text{iarray} for creating a new array.
  
  new () returns a new array of indefinite size; all elements are initialized as 0.

- **access**: \text{iarray} \rightarrow \text{int} \rightarrow \text{int} for accessing an array.
  
  access \ a \ \ i \ returns \ the \ i-th \ element \ of \ array \ a.

- **update**: \text{iarray} \rightarrow \text{int} \rightarrow \text{int} \rightarrow \text{unit} for updating an array.
  
  update \ a \ \ i \ \ n \ updates \ the \ i-th \ element \ of \ array \ a \ with \ integer \ n.

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

\[
\text{new } = \lambda _{\_} : \text{unit} . \text{ref } \lambda i : \text{int} . 0 \\
\text{access } = \lambda a : \text{iarray} . \lambda i : \text{int} . (\text{!} a) \ i \\
\text{update } = \lambda a : \text{iarray} . \lambda i : \text{int} . \lambda n : \text{int} .
\]

---

2
**Question 2. [10 pts]** State progress and type preservation theorems. In your statements, use the following judgments:

- A typing judgment \( \Gamma | \Psi \vdash e : A \) means that expression \( e \) has type \( A \) under typing context \( \Gamma \) and store typing context \( \Psi \).
- A reduction judgment \( e | \psi \mapsto e' | \psi' \) 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 \( \cdot | \Psi \vdash e : A \) for some store typing context \( \Psi \) and type \( A \). Then either:*

\[(1) \quad \text{____________________, or} \]

\[(2) \quad \text{for any __________________ such that __________________,}\]

\[\text{there exist some __________________ and __________________ such that __________________.}\]

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

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

\[\text{________________________} \quad \text{.}\]

\[\text{________________________} \quad \text{.}\]
2 Evaluation context and environment [28 pts]

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

\[
\begin{align*}
\text{type} & \quad A ::= P | A \to A \\
\text{base type} & \quad P ::= \text{bool} \\
\text{expression} & \quad e ::= x | \lambda x : A. e | e \ e | \text{true} | \text{false} | \text{if} \ e \ \text{then} \ e \ \text{else} \ e
\end{align*}
\]

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

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

**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:

environment $\eta ::= \cdot \mid \eta, x \mapsto v$

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

$$\eta \vdash e \mapsto v \iff e \text{ evaluates to } v \text{ under environment } \eta$$

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

$$\text{value } v ::= \ldots$$

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

$$\eta \vdash x \mapsto$$

$$\eta \vdash \lambda x : A. e \mapsto$$

$$\eta \vdash e_1 e_2 \mapsto$$
3 Subtyping [16 pts]


\[
\begin{align*}
A \rightarrow B & \leq A' \rightarrow B' & \text{Fun}_\leq \\
\text{ref } A & \leq \text{ref } B & \text{Ref}_\leq
\end{align*}
\]

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

\[
\frac{A \leq B}{\text{array } A \leq \text{array } B} \space Array_\leq'
\]

While it is controversial whether the rule \( Array_\leq' \) is a flaw in the design of the Java language, using the rule \( Array_\leq' \) 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 λ-calculus extended with recursive types:

<table>
<thead>
<tr>
<th>type</th>
<th></th>
<th>expression</th>
<th></th>
<th>typing context</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A ::= \text{unit} \mid A \rightarrow A \mid A + A \mid \alpha \mid \mu \alpha.A$</td>
<td></td>
<td>$e ::= x \mid \lambda x : A. e \mid e e \mid \text{inl}_A e \mid \text{inr}_A e \mid \text{case} e \text{ of } \text{inl} x. e \mid \text{inr} y. e \mid \text{fold}_C e \mid \text{unfold}_C e$</td>
<td>$\Gamma ::= \cdot \mid \Gamma, x : A \mid \Gamma, \alpha$ type</td>
<td></td>
</tr>
</tbody>
</table>

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

\[
\begin{array}{c}
C = \mu \alpha.A \\
\Gamma \vdash C \text{ type} \\
\hline
\Gamma \vdash \text{fold}_C e : \text{Fold}
\end{array}
\]

\[
\begin{array}{c}
C = \mu \alpha.A \\
\Gamma \vdash \text{unfold}_C e : \text{Unfold}
\end{array}
\]

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

\[
\begin{array}{c}
\text{datatype} \ \text{nat} = \text{Zero} \mid \text{Succ} \ of \ \text{nat}
\end{array}
\]

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

\[
\begin{array}{c}
\text{Zero} = \hphantom{\text{Zero}} \\
\text{Succ} e = \hphantom{\text{Succ} e}
\end{array}
\]
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 \( \Omega \to \Omega \) as well as type \( \Omega \). Or, in order for \( e_1 e_2 \) to be assigned type \( \Omega \), \( e_1 \) must be assigned \textit{not only} type \( \Omega \) but also type \( \Omega \to \Omega \) because \( e_2 \) is assigned type \( \Omega \). Thus \( \Omega \) must be identified with \( \Omega \to \Omega \).

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

\[
\begin{align*}
\Omega &= \quad \text{______________________________} \\
\quad x^\circ &= x \\
(\lambda x. e)^\circ &= \quad \text{______________________________} \\
(\lambda x. e)^\circ &= \quad \text{______________________________}
\end{align*}
\]
5 Polymorphism (36 pts)

The following shows the abstract syntax for System F:

\[
\begin{align*}
\text{type} & \quad A ::= A \rightarrow A \mid \alpha \mid \forall \alpha. A \\
\text{expression} & \quad e ::= x \mid \lambda x : A. e \mid e \ e \mid \Lambda \alpha. e \mid e \ [A]
\end{align*}
\]

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

\[
\begin{align*}
\text{erase}(x) & = x \\
\text{erase}(\lambda x : A. e) & = \lambda x. \text{erase}(e) \\
\text{erase}(e_1 \ e_2) & = \text{erase}(e_1) \ \text{erase}(e_2) \\
\text{erase}(\Lambda \alpha. e) & = \text{erase}(e) \\
\text{erase}(e \ [A]) & = \text{erase}(e)
\end{align*}
\]

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

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

**Question 3. [6 pts]** A Church numeral \( \hat{n} \) 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 \( \text{nat} \) in System F as follows:

\[
\text{nat} = \forall \alpha. (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)
\]

Encode a zero \( \text{zero} \) of type \( \text{nat} \) and a successor function \( \text{succ} \) of type \( \text{nat} \rightarrow \text{nat} \):

\[
\begin{align*}
\text{zero} & = \\
\text{succ} & = 
\end{align*}
\]
The following shows the abstract syntax for the let-polymorphism system:

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

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

expression \( e \) ::= \( x \) | \( \lambda x: A. e \) | \( e e \) | \( \Lambda \alpha. e \) | \( e [A] \) | \( \text{let } x: U = e \text{ in } e \)

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:

\[
\begin{align*}
erase(x) &= x \\
erase(\lambda x: A. e) &= \lambda x. erase(e) \\
erase(e_1 e_2) &= erase(e_1) \ erase(e_2) \\
erase(\Lambda \alpha. e) &= erase(e) \\
erase(e [A]) &= erase(e) \\
erase(\text{let } x: U = e \text{ in } e') &= \text{let } x = erase(e) \text{ in } erase(e')
\end{align*}
\]

**Question 4.** [5 pts] Give a well-typed closed expression \( e \) in the let-polymorphism system such that \( erase(e) = \text{let } f = \lambda x. x \text{ in } (f \ true, f \ 0) \). Assume two monotypes \( \text{bool} \) for boolean values and \( \text{int} \) 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 \( erase(e) = \text{let } f = (\lambda x. x) (\lambda y. y) \text{ in } (f \ true, f \ 1) \). 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.

\[
\begin{align*}
\text{monotype} & \quad A ::= A \rightarrow A \mid \alpha \\
\text{polytype} & \quad U ::= A \mid \forall \alpha. U \\
\text{expression} & \quad e ::= x \mid \lambda x. e \mid e\ e \mid \text{let } x = e \text{ in } e \\
\text{typing context} & \quad \Gamma ::= \cdot \mid \Gamma, x : U \\
\text{type substitution} & \quad S ::= \text{id} \mid \{ A/\alpha \} \mid S \circ S \\
\text{type equations} & \quad E ::= \cdot \mid E, A = A
\end{align*}
\]

- $S \cdot U$ 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 $W$, 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 $W$. Use the typing judgment of the form $\Gamma \vdash x : U$ in your statement.

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

then ____________________________________________________________________________________.
Question 2. [14 pts] Assume that you are given the unification algorithm \texttt{Unify}(E) and an auxiliary function \texttt{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:

\[
\begin{align*}
\mathcal{W}(\Gamma, x) & = (\text{id}, \{\vec{\beta}/\vec{\alpha}\} \cdot A) & x : \forall \vec{\alpha}. A \in \Gamma \text{ and fresh } \vec{\beta} \\
\mathcal{W}(\Gamma, \lambda x. e) & = \text{let } (S, A) = \mathcal{W}(\Gamma + x : \alpha, e) \text{ in } \text{fresh } \alpha \\
\mathcal{W}(\Gamma, e_1 e_2) & = \text{let } (S_1, A_1) = \mathcal{W}(\Gamma, e_1) \text{ in } \text{let } \\
\mathcal{W}(\Gamma, \text{let } x = e_1 \text{ in } e_2) & = \text{let } \\
\end{align*}
\]
Question 3. [6 pts] Given an application $e_1 \ e_2$, the algorithm $W$ reconstructs first the type of $e_1$ and then the type of $e_2$. Modify the algorithm $W$ so that it reconstructs first the type of $e_2$ and then the type of $e_1$.

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

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:

$$\frac{}{\Gamma \vdash (e_1, e_2) : A_1 \times A_2} \times_l$$

Complete the case for $(e_1, e_2)$ in the algorithm $W$:

$$W(\Gamma, (e_1, e_2)) = \text{___________________________________________________________________________}$$

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

Substitution =  \\

Type =  \\

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