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.

\[ \text{PL} \ 2010^\lambda \ [\text{Extracredit}] \]

State “Yes” if you wear the \text{PL} \ 2010^\lambda T-shirt.

\[ \text{PL} \ 2010 \text{ Tekken Match} \ [\text{Extracredit}] \]

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.

\[
\begin{align*}
type & \quad \mathcal{A} ::= \mathcal{P} \mid \mathcal{A} \to \mathcal{A} \mid \text{int} \mid \text{ref } \mathcal{A} \\
expression & \quad \mathcal{e} ::= x \mid \lambda x: \mathcal{A}. \mathcal{e} \mid \mathcal{e} \mathcal{e} \mid \text{let } x = \mathcal{e} \text{ in } \mathcal{e} \mid \text{ref } \mathcal{e} \mid \!\mathcal{e} \mid \mathcal{e} ::= \mathcal{e} \mid 0 \mid 1 \mid \cdots \\
value & \quad \mathcal{v} ::= \lambda x: \mathcal{A}. \mathcal{e} \mid \mathcal{l} \mid \mathcal{v} \mid \cdots \\
store & \quad \psi ::= \cdot \mid \psi, l \mapsto \mathcal{v} \\
typing context & \quad \Gamma ::= \cdot \mid \Gamma, x : \mathcal{A} \\
store typing context & \quad \Psi ::= \cdot \mid \Psi, l \mapsto \mathcal{A}
\end{align*}
\]

**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 element of the array. We choose a functional representation of arrays by defining type iarray for arrays of integers as follows:

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

We need the following constructs for arrays:

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

- **access** : iarray \(\to\) int \(\to\) int for accessing an array.
  access \(a\) \(i\) returns the \(i\)-th element of array \(a\).

- **update** : iarray \(\to\) int \(\to\) int \(\to\) 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:

\[
\begin{align*}
\text{new} & = \lambda_-: \text{unit}. \text{ref } \lambda i: \text{int}. 0 \\
\text{access} & = \lambda a: \text{iarray}. \lambda i: \text{int}. (\lambda a) \ i \\
\text{update} & = \lambda a: \text{iarray}. \lambda i: \text{int}. \lambda n: \text{int}.
\end{align*}
\]

\[
\text{let } old = a \text{ in} \\
\text{a} ::= \lambda j: \text{int}. \text{if } i = j \text{ then } n \text{ else } old \ j
\]

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. $e$ is a value ________, or
2. for any store $\psi$ ________ such that $\psi :: \Psi$ __________, there exist some expression $e'$ __________ and store $\psi'$ __________ such that $e | \psi \mapsto e' | \psi'$ __________.

**Theorem (Type preservation).** Suppose \[
\begin{cases}
\Gamma | \Psi \vdash e : A \\
\psi :: \Psi \\
e | \psi \mapsto e' | \psi'
\end{cases}
\]
Then there exists a store typing context $\Psi'$ such that

\[
\begin{cases}
\Gamma | \Psi' \vdash e' : A \\
\Psi \subset \Psi' \\
\psi' :: \Psi'
\end{cases}
\]
2 Evaluation context and environment [28 pts]

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

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

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

\[
\text{evaluation context } \kappa ::= \square \mid \kappa \; e \mid (\lambda x : A. e) \; \kappa \mid \text{if } \kappa \text{ then } e \text{ else } e
\]

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

\[
\text{evaluation context } \kappa ::= \square \mid \kappa \; e \mid \text{if } \kappa \text{ then } e \text{ else } e
\]

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.

\[(\lambda y : \text{bool}. (\lambda x : \text{bool}. x) \; y) \; \text{true}\]

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 $\kappa \neq \kappa'$.

\[(\lambda x : \text{bool} \rightarrow \text{bool}. x) \; (\lambda y : \text{bool}. y) \; \text{true}\]
Question 5. [8 pts] The key idea behind the environment semantics is to postpone a substitution \([v/x]\) by storing a pair of value \(v\) and variable \(x\) in an environment. We use the following definition of environment:

\[
\text{environment } \eta ::= \cdot \mid \eta, x \mapsto v
\]

\(\cdot\) 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.

\[
\text{value } v ::= [\eta, \lambda x : A. e] \mid \text{true} \mid \text{false}
\]

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

\[
\begin{align*}
\frac{x \mapsto v \in \eta}{\eta \vdash x \mapsto v} \\

\frac{\eta \vdash \lambda x : A. e \mapsto [\eta, \lambda x : A. e]}{}
\end{align*}
\]

\[
\begin{align*}
\frac{\eta \vdash e_1 \mapsto [\eta', \lambda x : A. e] \quad \eta \vdash e_2 \mapsto v' \quad \eta', x \mapsto v' \vdash e \mapsto v}{\eta \vdash e_1 \ e_2 \mapsto v}
\end{align*}
\]
3 Subtyping [16 pts]


\[
A' \leq A \quad B \leq B' \\
A \rightarrow B \leq A' \rightarrow B' \quad \text{Fun}_\leq
\]

\[
A \leq B \\
\text{Ref}_\leq
\]

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

\[
A \leq B \\
\text{array}_\leq
\]

While it is controversial whether the rule \textit{Array}_\leq is a flaw in the design of the Java language, using the rule \textit{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.

\textit{Answer:} Whenever a value of type \( B \) is written to an array of type \( \text{array}_A \), the runtime system must verify a subtyping relation \( B \leq A \), which incurs a runtime overhead of dynamic tag-checks.
4 Recursive types [12 pts]

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

\[
\begin{align*}
\text{type} & \quad A ::= \text{unit} | A \rightarrow A | A + A | \alpha | \mu \alpha. A \\
\text{expression} & \quad e ::= x | \lambda x : A. e | e e | \\
& \quad \text{inl}_A e | \text{inr}_A e | \text{case } e \text{ of } \text{inl } x. e | \text{inr } y. e | \\
& \quad \text{fold}_C e | \text{unfold}_C e \\
\text{typing context} & \quad \Gamma ::= \cdot | \Gamma, x : A | \Gamma, \alpha \text{ type}
\end{align*}
\]

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

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

**Fold**

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

**Unfold**

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

\[
\text{datatype } \text{nat} = \text{Zero} | \text{Succ } \text{of } \text{nat}
\]

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

\[
\begin{align*}
\text{Zero} & = \text{fold}_\text{nat} \text{ inl}_\text{nat} () \\
\text{Succ } e & = \text{fold}_\text{nat} \text{ inr}_\text{unit} e
\end{align*}
\]
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:

\[
\begin{align*}
\Omega &= \mu\alpha.\alpha \rightarrow \alpha \\
x^\circ &= x \\
(\lambda x. e)^\circ &= \text{fold}_\Omega \lambda x:\Omega. e^\circ \\
(e_1 e_2)^\circ &= (\text{unfold}_\Omega e_1^\circ) e_2^\circ
\end{align*}
\]
5 Polymorphism (36 pts)

The following shows the abstract syntax for System F:

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

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.

\[
\lambda x : \forall \alpha.\alpha \to \alpha.\ x\ [\forall \alpha.\alpha \to \alpha] \ x
\]

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.

Answer: There is no such expression.

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:

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

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

\[
\begin{align*}
\text{zero} &= \Lambda\alpha.\ \lambda f : \alpha \to \alpha.\ \lambda x : \alpha.\ x \\
\text{succ} &= \lambda n : \text{nat}.\ \Lambda\alpha.\ \lambda f : \alpha \to \alpha.\ \lambda x : \alpha.\ (n\ [\alpha] \ f)\ (f\ x)
\end{align*}
\]
The following shows the abstract syntax for the let-polymorphism system:

\[
\begin{align*}
\text{monotype} & \quad \text{A} & ::= & \quad \text{A} \to \text{A} \mid \alpha \\
\text{polytype} & \quad \text{U} & ::= & \quad \text{A} \mid \forall \alpha. \text{U} \\
\text{expression} & \quad \text{e} & ::= & \quad \text{x} \mid \lambda x: \text{A}. \text{e} \mid \text{e} \text{ e} \mid \Lambda \alpha. \text{e} \mid \text{e} \llbracket \text{A} \rrbracket \mid \text{let } x: \text{U} = \text{e} \text{ in } \text{e}
\end{align*}
\]

Below we define an erasure function \( \text{erase}(\cdot) \) 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:

\[
\begin{align*}
\text{erase}(x) & = x \\
\text{erase}(\lambda x: \text{A}. \text{e}) & = \lambda x. \text{erase(e)} \\
\text{erase}(\text{e}_1 \text{ e}_2) & = \text{erase(e}_1) \text{ erase(e}_2) \\
\text{erase}(\Lambda \alpha. \text{e}) & = \text{erase(e)} \\
\text{erase}(\text{e} \llbracket A \rrbracket) & = \text{erase(e)} \\
\text{erase(let } x: \text{U} = \text{e} \text{ in } \text{e}') & = \text{let } x = \text{erase(e)} \text{ in } \text{erase(e)'}
\end{align*}
\]

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

\[
\text{let } f: \forall \alpha. \alpha \to \alpha = \Lambda \alpha. \lambda x: \alpha. x \text{ in } (f \llbracket \text{bool} \rrbracket \text{ true}, f \llbracket \text{int} \rrbracket \text{ 0})
\]

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

*Answer:* Value restriction allows variable \( x \) in a let-binding \( \text{let } x = \text{e} \text{ in } \text{e}' \) to be assigned a polytype only if expression \( \text{e} \) is a value.

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

*Answer:* Due to the value restriction, \( f \) cannot have a polytype such as \( \forall \alpha. \alpha \to \alpha \). Therefore \( f \) cannot be applied to two values \( \text{true} \) and \( 1 \) of different types at the same time.
6 Type reconstruction [40 pts]

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

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

- \(S \cdot U \) 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 \(\Gamma \triangleright x : U\) in your statement.

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

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

\[
\begin{align*}
W(\Gamma, x) &= (\text{id}, \{ \vec{\beta}/\vec{\alpha} \} \cdot A) & x : \forall \vec{\alpha}. A \in \Gamma \text{ and fresh } \vec{\beta} \\
W(\Gamma, \lambda x. e) &= \text{let } (S, A) = W(\Gamma + x : \alpha, e) \text{ in } \text{fresh } \alpha \\
W(\Gamma, e_1 e_2) &= \text{let } (S_1, A_1) = W(\Gamma, e_1) \text{ in } \\
&\hspace{1cm} \text{let } (S_2, A_2) = W(S_1 \cdot \Gamma, e_2) \text{ in } \\
&\hspace{1cm} \text{let } S_3 = \text{Unify}(S_2 \cdot A_1 = A_2 \rightarrow \alpha) \text{ in } \text{fresh } \alpha \\
&\hspace{1cm} (S_3 \circ S_2 \circ S_1, S_3 \cdot \alpha) \\
W(\Gamma, \text{let } x = e_1 \text{ in } e_2) &= \text{let } (S_1, A_1) = W(\Gamma, e_1) \text{ in } \\
&\hspace{1cm} \text{let } (S_2, A_2) = W(S_1 \cdot \Gamma + x : \text{Gen}_{S_1, \Gamma}(A_1), e_2) \text{ in } \\
&\hspace{1cm} (S_2 \circ S_1, A_2)
\end{align*}
\]
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 \( e_1 \).

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

\[
\text{let } (S_1, A_1) = W(S_2 \cdot \Gamma, e_1) \text{ in }
\]

\[
\text{let } S_3 = \text{Unify}(A_1 = (S_1 \cdot A_2) \rightarrow \alpha) \text{ in fresh } \alpha
\]

\[(S_3 \circ S_1 \circ S_2, S_3 \cdot \alpha)\]

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:

\[
\frac{\Gamma \triangleright e_1 : A_1 \quad \Gamma \triangleright e_2 : A_2}{\Gamma \triangleright (e_1, e_2) : A_1 \times A_2} \times 1
\]

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

\[
W(\Gamma, (e_1, e_2)) = \text{let } (S_1, A_1) = W(\Gamma, e_1) \text{ in }
\]

\[
\text{let } (S_2, A_2) = W(S_1 \cdot \Gamma, e_2) \text{ in }
\]

\[(S_2 \circ S_1, (S_2 \cdot A_1) \times A_2)\]
Question 5. [8 pts] What is the result of \(W(\cdot, \text{let } f = \lambda x. x \text{ in } (f \ 0, f \ \text{true}))\)? Assume two monotypes bool for boolean values and int for integers.

Substitution \(= \{\text{int}/\alpha_4\} \circ \{\text{int}/\alpha_3\} \circ \{\text{bool}/\alpha_2\} \circ \{\text{bool}/\alpha_1\}\)

Type \(= \text{int} \times \text{bool}\)

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

1. \(\alpha_1\) is generated when the algorithm \(W\) specializes the polymorphic type of \(f\) for \(f \ 0\);
2. \(\alpha_2\) when \(W\) reconstructs the type of \(f \ 0\);
3. \(\alpha_3\) when \(W\) specializes the polymorphic type of \(f\) for \(f \ \text{true}\); and
4. \(\alpha_4\) when \(W\) reconstructs the type of \(f \ \text{true}\).
Work sheet