

Name:

Hemos ID:

CSE-321 Programming Languages 2010
Final — Sample Solution

	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^λ [Extracredit]

State “Yes” if you wear the PL 2010^λ 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.

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

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 `iarray` for arrays of integers as follows:

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

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 -th element of array a with integer n .

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

$$\text{new} = \lambda_.:\text{unit}. \text{ref } \lambda i:\text{int}. 0$$

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

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

let $old = !a$ in

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

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 Γ and store typing context Ψ .
- A reduction judgment $e \mid \psi \mapsto e' \mid \psi'$ means that e with store ψ reduces to e' with ψ' .
- A store judgment $\psi :: \Psi$ means that Ψ corresponds to ψ .

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) *e is a value _____, or*

(2) *for any store ψ _____ such that $\psi :: \Psi$ _____,*

there exist some expression e' _____ and store ψ' _____ such that $e \mid \psi \mapsto e' \mid \psi'$ _____.

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 Ψ' such that $\left\{ \begin{array}{l} \Gamma \mid \Psi' \vdash e' : A \\ \Psi \subset \Psi' \\ \psi' :: \Psi' \end{array} \right.$.

2 Evaluation context and environment [28 pts]

Consider the following fragment of the simply-typed λ -calculus.

type	$A ::= P \mid A \rightarrow A$
base type	$P ::= \text{bool}$
expression	$e ::= x \mid \lambda x:A. e \mid e e \mid \text{true} \mid \text{false} \mid \text{if } e \text{ then } e \text{ else } e$

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

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.

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 $[v/x]e$ 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 $\eta \vdash e \mapsto v$:

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

Give the definition of values for the simply-typed λ -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.

$$\frac{x \mapsto v \in \eta}{\eta \vdash x \mapsto v}$$

$$\overline{\eta \vdash \lambda x:A. e \mapsto [\eta, \lambda x:A. e]}$$

$$\frac{\eta \vdash e_1 \mapsto [\eta', \lambda x:A. e] \quad \eta \vdash e_2 \mapsto v' \quad \eta', x \mapsto v' \vdash e \mapsto v}{\eta \vdash e_1 e_2 \mapsto v}$$

3 Subtyping [16 pts]

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

$$\frac{A' \leq A \quad B \leq B'}{A \rightarrow B \leq A' \rightarrow B'} \text{Fun}_{\leq}$$

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

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

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

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.

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 λ -calculus extended with recursive types:

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

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

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

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

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

$$\text{datatype nat} = \text{Zero} \mid \text{Succ of nat}$$

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

$$\text{Zero} = \text{fold}_{\text{nat}} \text{inl}_{\text{nat}} ()$$

$$\text{Succ } e = \text{fold}_{\text{nat}} \text{inr}_{\text{unit}} e$$

Question 3. [Extracredit] We want to translate an expression e in the untyped λ -calculus into an expression e° in the simply typed λ -calculus extended with recursive types. 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 $\Omega \rightarrow \Omega$ *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 $\Omega \rightarrow \Omega$ because e_2 is assigned type Ω . Thus Ω must be identified with $\Omega \rightarrow \Omega$.

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

$$\Omega = \mu\alpha. \alpha \rightarrow \alpha$$

$$x^\circ = x$$

$$(\lambda x. e)^\circ = \text{fold}_\Omega \lambda x : \Omega. e^\circ$$

$$(e_1 e_2)^\circ = (\text{unfold}_\Omega e_1^\circ) e_2^\circ$$

5 Polymorphism (36 pts)

The following shows the abstract syntax for System F:

$$\begin{array}{ll} \text{type} & A ::= A \rightarrow A \mid \alpha \mid \forall \alpha. A \\ \text{expression} & e ::= x \mid \lambda x : A. e \mid e e \mid \Lambda \alpha. e \mid e \llbracket A \rrbracket \end{array}$$

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 λ -calculus:

$$\begin{array}{ll} erase(x) & = x \\ erase(\lambda x : A. e) & = \lambda x. erase(e) \\ erase(e_1 e_2) & = erase(e_1) erase(e_2) \\ erase(\Lambda \alpha. e) & = erase(e) \\ erase(e \llbracket A \rrbracket) & = erase(e) \end{array}$$

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.

$$\lambda x : \forall \alpha. \alpha \rightarrow \alpha. x \llbracket \forall \alpha. \alpha \rightarrow \alpha \rrbracket x$$

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.

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 nat in System F as follows:

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

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

$$\text{zero} = \Lambda \alpha. \lambda f : \alpha \rightarrow \alpha. \lambda x : \alpha. x$$

$$\text{succ} = \lambda n : \text{nat}. \Lambda \alpha. \lambda f : \alpha \rightarrow \alpha. \lambda x : \alpha. (n \llbracket \alpha \rrbracket f) (f x)$$

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

monotype	$A ::= A \rightarrow A \mid \alpha$
polytype	$U ::= A \mid \forall \alpha. U$
expression	$e ::= x \mid \lambda x:A. e \mid e e \mid \Lambda \alpha. e \mid e \llbracket A \rrbracket \mid \text{let } x:U = e \text{ in } e$

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:

$$\begin{aligned}
erase(x) &= x \\
erase(\lambda x:A. e) &= \lambda x. erase(e) \\
erase(e_1 e_2) &= erase(e_1) erase(e_2) \\
erase(\Lambda \alpha. e) &= erase(e) \\
erase(e \llbracket A \rrbracket) &= erase(e) \\
erase(\text{let } x:U = e \text{ in } e') &= \text{let } x = erase(e) \text{ in } erase(e')
\end{aligned}$$

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

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

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

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

Question 6. [5 pts] Give a well-typed closed expression e in the let-polymorphism system with value restriction such that $erase(e) = \text{let } f = (\lambda x. x) (\lambda y. y) \text{ in } (f \text{ true}, f 1)$. 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 $\forall \alpha. \alpha \rightarrow \alpha$. Therefore f cannot be applied to two values `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.

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

- $S \cdot U$ and $S \cdot \Gamma$ denote applications of S to U and Γ , respectively.
- $ftv(\Gamma)$ denotes the set of free type variables in Γ ; $ftv(U)$ 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 Γ contains no type binding for variable x .

Question 1. [6 pts] The type reconstruction algorithm, called \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. 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 $S \cdot \Gamma \triangleright e : A$

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 Γ . The following specifies the algorithm \mathcal{W} . Complete the specification:

$$\begin{aligned} \mathcal{W}(\Gamma, x) &= (\text{id}, \{\vec{\beta}/\vec{\alpha}\} \cdot A) && x : \forall \vec{\alpha}. A \in \Gamma \text{ and fresh } \vec{\beta} \\ \mathcal{W}(\Gamma, \lambda x. e) &= \text{let } (S, A) = \mathcal{W}(\Gamma + x : \alpha, e) \text{ in} && \text{fresh } \alpha \end{aligned}$$

$$(S, (S \cdot \alpha) \rightarrow A)$$

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

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

$$\text{let } S_3 = \text{Unify}(S_2 \cdot A_1 = A_2 \rightarrow \alpha) \text{ in} \quad \text{fresh } \alpha$$

$$(S_3 \circ S_2 \circ S_1, S_3 \cdot \alpha)$$

$$\mathcal{W}(\Gamma, \text{let } x = e_1 \text{ in } e_2) = \text{let } (S_1, A_1) = \mathcal{W}(\Gamma, e_1) \text{ in}$$

$$\text{let } (S_2, A_2) = \mathcal{W}(S_1 \cdot \Gamma + x : \text{Gen}_{S_1 \cdot \Gamma}(A_1), e_2) \text{ in}$$

$$(S_2 \circ S_1, A_2)$$

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 .

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

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

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

$$(S_3 \circ S_1 \circ S_2, S_3 \cdot \alpha)$$

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:

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

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

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

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

$$(S_2 \circ S_1, (S_2 \cdot A_1) \times A_2)$$

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 `bool` for boolean values and `int` for integers.

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

$$\text{Type} = \text{int} \times \text{bool}$$

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

1. α_1 is generated when the algorithm \mathcal{W} specializes the polymorphic type of f for $f\ 0$;
2. α_2 when \mathcal{W} reconstructs the type of $f\ 0$;
3. α_3 when \mathcal{W} specializes the polymorphic type of f for $f\ \text{true}$; and
4. α_4 when \mathcal{W} reconstructs the type of $f\ \text{true}$.

Work sheet

Work sheet