

Name:

Hemos ID:

CSE-321 Programming Languages 2009  
Final — Sample Solution

	Prob 1	Prob 2	Prob 3	Prob 4	Prob 5	Prob 6	Prob 7	Total
Score								
Max	10	10	30	25	30	15	40	160

- There are seven problems on 14 pages, including a work sheet, in this exam.
- The maximum score for this exam is 160 points.
- Be sure to write your name and Hemos ID.
- You have one and half 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.

---

# 1 Simply-typed $\lambda$ -calculus[10 pts]

Consider the following SML program:

```
fun f 0 = 1
  | f x = x + (g (x - 1))
and g 0 = 1
  | g x = x - (f (x - 1))
```

The function `f` calls the function `g`, and the function `g` calls the function `f`. We refer to these functions as mutually recursive functions.

The goal of this problem is to rewrite an SML expression `f 10` in the following fragment of simply-typed  $\lambda$ -calculus:

type	$A ::= \text{int} \mid \text{bool} \mid A \rightarrow A \mid A \times A$
expression	$e ::= x \mid \lambda x:A. e \mid e e \mid (e, e) \mid \text{fst } e \mid \text{snd } e \mid () \mid$ $\text{true} \mid \text{false} \mid \text{if } e \text{ then } e \text{ else } e \mid \text{fix } x:A. e$ $0 \mid 1 \mid 2 \mid \dots$

For the sake of simplicity, we assume that the infix operations  $+$ ,  $-$  and  $=$  are given as primitive, which correspond to integer addition, integer subtraction and integer comparison, respectively. Write an expression that corresponds to `f 10`.

$(\lambda fg:\text{int} \rightarrow \text{int} \times \text{int} \rightarrow \text{int}. \text{fst } f \ 10)$

$(\text{fix } fg:\text{int} \rightarrow \text{int} \times \text{int} \rightarrow \text{int}.$

$(\lambda x:\text{int}. \text{if } x = 0 \text{ then } 1 \text{ else } x + (\text{snd } fg \ (x - 1))),$

$\lambda x:\text{int}. \text{if } x = 0 \text{ then } 1 \text{ else } x - (\text{fst } fg \ (x - 1))))$

## 2 Mutable references [10 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 construct for mutable references to implement `new`, `access` and `update`. Fill in the blank:

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

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

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

$$\text{let } old = !a \text{ in}$$

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

### 3 Evaluation context and environment [30 pts]

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

type	$A ::= \text{unit} \mid A \rightarrow A$
expression	$e ::= x \mid \lambda x:A. e \mid e e \mid ()$
value	$v ::= \lambda x:A. e \mid ()$

In this problem, we will consider various operational semantics based on the call-by-value (CBV) strategy.

**Question 1.** [5 pts] Give the rules for the reduction judgment  $e \mapsto e'$ .

$$\frac{e_1 \mapsto e'_1}{e_1 e_2 \mapsto e'_1 e_2}$$

$$\frac{e_2 \mapsto e'_2}{v e_2 \mapsto v e'_2}$$

$$\overline{(\lambda x:A. e) v \mapsto [v/x]e}$$

**Question 2.** [5 pts] Assuming that the reduction relation  $\mapsto_\beta$  denotes the  $\beta$ -reduction, the following reduction rule alone specifies a reduction strategy completely because the order of reduction is implicitly determined by the definition of evaluation contexts:

$$\frac{e \mapsto_\beta e'}{\kappa[e] \mapsto \kappa[e']} \text{Red}_\beta$$

Give the definition of evaluation context which leads to the CBV strategy.

$$\text{evaluation context} \quad \kappa ::= \square \mid \kappa e \mid (\lambda x:A. e) \kappa$$

**Question 3. [5 pts]** There is another judgment which differs from the reduction judgment called *evaluation judgment* of the form  $e \hookrightarrow v$ :

$$e \hookrightarrow v \quad \Leftrightarrow \quad e \text{ evaluates to } v$$

It takes a single *big* step with which we immediately finish evaluating a given expression. Give the rules for the evaluation judgment  $e \hookrightarrow v$  corresponding to the CBV strategy.

$$\overline{v \hookrightarrow v}$$

$$\frac{e_1 \hookrightarrow \lambda x:A. e \quad e_2 \hookrightarrow v_2 \quad [v_2/x]e \hookrightarrow v}{e_1 \ e_2 \hookrightarrow v}$$

**Question 4. [5 pts]** The key idea behind the environment semantics is to postpone a substitution  $[v/x]e$  by storing a pair of  $v$  and  $x$  in an *environment*. We use the following definition of environment:

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

$\cdot$  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  $\eta \vdash e \hookrightarrow v$ :

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

Complete the rules for the environment evaluation judgment  $\eta \vdash e \hookrightarrow v$  corresponding to the CBV strategy.

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

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

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

**Question 5.** [10 pts] The abstract machine  $\mathbf{E}$  is for a practical implementation of the environment semantics. There are two kinds of states in the abstract machine  $\mathbf{E}$ .

- $\sigma \blacktriangleright e @ \eta$  means that the machine is currently analyzing  $e$  under the environment  $\eta$ . In order to evaluate a variable in  $e$ , we look up the environment  $\eta$ .
- $\sigma \blacktriangleleft v$  means that the machine is currently returning  $v$  to the stack  $\sigma$ . We do not need an environment for  $v$  because the evaluation of  $v$  has been finished.

The following shows the definition of the abstract machine  $\mathbf{E}$ . Complete the definition.

value	$v ::= [\eta, \lambda x:A. e]$
environment	$\eta ::= \cdot \mid \eta, x \hookrightarrow v$
frame	$\phi ::= \Box_\eta e \mid [\eta, \lambda x:A. e] \Box$
stack	$\sigma ::= \Box \mid \sigma; \phi$
state	$s ::= \sigma \blacktriangleright e @ \eta \mid \sigma \blacktriangleleft v$

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

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

$$\frac{}{\sigma \blacktriangleright e_1 e_2 @ \eta \mapsto_{\mathbf{E}} \sigma; \Box_\eta e_2 \blacktriangleright e_1 @ \eta}$$

$$\frac{}{\sigma; \Box_\eta e_2 \blacktriangleleft [\eta', \lambda x:A. e] \mapsto_{\mathbf{E}} \sigma; [\eta', \lambda x:A. e] \Box \blacktriangleright e_2 @ \eta}$$

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

## 4 Subtyping [25 pts]

**Question 1.** [10 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 whenever 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. Give the two subtyping rules:

$$\frac{}{A \leq A} \text{Ref}_{\leq}$$

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

**Question 2.** [10 pts] The rule of *subsumption* is a typing rule which enables us to change the type of an expression to its subtype. Complete the rule of subsumption:

$$\frac{\Gamma \vdash e : A \quad A \leq B}{\Gamma \vdash e : B} \text{Sub}$$

**Question 3.** [5 pts] Complete the subtyping rule for product types, function types and reference types.

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

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

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

## 5 Recursive types [30 pts]

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

type	$A ::= A \rightarrow A \mid \alpha \mid \mu\alpha.A$
expression	$e ::= x \mid \lambda x:A. e \mid e e \mid \text{fold}_C e \mid \text{unfold}_C e$
typing context	$\Gamma ::= \cdot \mid \Gamma, x:A \mid \Gamma, \alpha \text{ type}$

**Question 1.** [5 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.** [5 pts] Define the CBV operational semantics of those constructor for recursive types. Give reduction rules based on the call-by-value strategy. You have to write only those rules related with  $\text{fold}_C e$  and  $\text{unfold}_C e$ .

$$\frac{e \mapsto e'}{\text{fold}_C e \mapsto \text{fold}_C e'} \text{ Fold}$$

$$\frac{e \mapsto e'}{\text{unfold}_C e \mapsto \text{fold}_C e'} \text{ Unfold}$$

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



**Question 3. [10 pts]** Translate a recursive datatype for lists of natural numbers into the simply-typed  $\lambda$ -calculus extended with recursive types.

$$\text{datatype nlist} = \text{Nil} \mid \text{Cons of nat} \times \text{nlist}$$

$$\text{nlist} = \mu\alpha. \text{unit} + (\text{nat} \times \alpha)$$

$$\text{Nil} = \text{fold}_{\text{nlist}} \text{inl}_{\text{nat} \times \text{nlist}} ()$$

$$\text{Cons } e = \text{fold}_{\text{nlist}} \text{inr}_{\text{unit}} e$$

$$\text{case } e \text{ of Nil} \Rightarrow e_1 \mid \text{Cons } x \Rightarrow e_2 = \text{case unfold}_{\text{nlist}} e \text{ of inl } \_ . e_1 \mid \text{inr } x . e_2$$

**Question 4. [10 pts]** 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:

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

$$x^\circ = x$$

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

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

## 6 Polymorphism (15 pts)

There are three type systems for polymorphism: System F, Predicative Polymorphic  $\lambda$ -calculus and Let-polymorphism. In this problem, we will examine the definition of each type system.

**Question 1. [5 pts]** The following shows the definition of System F. Complete the definition:

type	$A ::= A \rightarrow A \mid \alpha \mid \forall \alpha. A$
expression	$e ::= x \mid \lambda x : A. e \mid e e \mid \Lambda \alpha. e \mid e \llbracket A \rrbracket$
typing context	$\Gamma ::= \cdot \mid \Gamma, \alpha \text{ type} \mid \Gamma, x : A$

**Question 2. [5 pts]** The following shows the definition of Predicative Polymorphic  $\lambda$ -calculus. Complete the definition:

monotype	$A ::= A \rightarrow A \mid \alpha$
polytype	$U ::= A \mid \forall \alpha. U$
expression	$e ::= x \mid \lambda x : A. e \mid e e \mid \Lambda \alpha. e \mid e \llbracket A \rrbracket$
typing context	$\Gamma ::= \cdot \mid \Gamma, \alpha \text{ type} \mid \Gamma, x : A$

**Question 3. [5 pts]** The following shows the definition of Let-polymorphism. Complete the definition.

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

## 7 Type reconstruction [40 pts]

In this problem, we will examine the type reconstruction algorithm, called  $\mathcal{W}$ , for the following expressions:

$$\text{expression} \quad e ::= x \mid \lambda x. e \mid e \ e \mid \text{let } x = e \text{ in } e$$

**Question 1. [5 pts]** A type substitution is a mapping from type variables to monotypes. The following defines the application of a type substitution. Complete the definition:

$$\begin{aligned} \text{id} \cdot U &= U \\ \{A/\alpha\} \cdot \alpha &= A \\ \{A/\alpha\} \cdot \beta &= \beta && \text{where } \alpha \neq \beta \\ \{A/\alpha\} \cdot B_1 \rightarrow B_2 &= \{A/\alpha\} B_1 \rightarrow \{A/\alpha\} B_2 \\ \{A/\alpha\} \cdot \forall \alpha. U &= \forall \alpha. U \\ \{A/\alpha\} \cdot \forall \beta. U &= \forall \beta. \{A/\alpha\} U && \text{where } \alpha \neq \beta \\ S_1 \circ S_2 \cdot U &= S_1 \cdot (S_2 \cdot U) \end{aligned}$$

**Question 2. [5 pts]**  $\text{Unify}(E)$  is a function which attempts to calculate a type substitution that unifies two types  $A$  and  $A'$  in each type equation  $A = A'$ . If no such type substitution exists,  $\text{Unify}(E)$  returns *fail*. The following shows the definition of  $\text{Unify}(E)$ . Complete the definition. You may use an auxiliary function  $\text{ftv}(A)$ , which denotes the set of free type variables in  $A$ :

$$\begin{aligned} \text{Unify}(\cdot) &= \text{id} \\ \text{Unify}(E, \alpha = A) = \text{Unify}(E, A = \alpha) &= \begin{aligned} &\text{if } \alpha = A \text{ then } \text{Unify}(E) \\ &\text{else if } \alpha \in \text{ftv}(A) \text{ then } \text{fail} \\ &\text{else } \text{Unify}(\{A/\alpha\} \cdot E) \circ \{A/\alpha\} \end{aligned} \\ \text{Unify}(E, A_1 \rightarrow A_2 = B_1 \rightarrow B_2) &= \text{Unify}(E, A_1 = B_1, A_2 = B_2) \end{aligned}$$

**Question 3. [5 pts]**  $\text{Gen}_\Gamma(A)$  is a function which generalizes monotype  $A$  to a polytype after taking into account free type variables in typing context  $\Gamma$ . Here are a few examples of  $\text{Gen}_\Gamma(A)$ . Fill in the blank:

$$\begin{aligned}\text{Gen}_\Gamma(\alpha \rightarrow \alpha) &= \forall \alpha. \alpha \rightarrow \alpha \\ \text{Gen}_{x:\alpha}(\alpha \rightarrow \alpha) &= \alpha \rightarrow \alpha \\ \text{Gen}_{x:\alpha}(\alpha \rightarrow \beta) &= \forall \beta. \alpha \rightarrow \beta \\ \text{Gen}_{x:\alpha, y:\beta}(\alpha \rightarrow \beta) &= \alpha \rightarrow \beta\end{aligned}$$

**Question 4. [15 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. The following specifies the algorithm  $\mathcal{W}$ . Complete the specification. You may use auxiliary functions  $\text{Unify}(E)$  and  $\text{Gen}_\Gamma(A)$ :

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

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

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

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

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

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

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

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

**Question 5. [10 pts]** Until now, we consider the algorithm  $\mathcal{W}$  for the following expressions:

$$\text{expression} \quad e ::= x \mid \lambda x. e \mid e e \mid \text{let } x = e \text{ in } e$$

The definitions of monotype  $A$ , polytype  $U$  and typing context  $\Gamma$  are the same with those in the Let-polymorphism. If the algorithm  $\mathcal{W}$  is correct, the result of  $\mathcal{W}(\Gamma, e)$  should be related with the above typing rules. We refer to this property as *soundness*. 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  $S \cdot \Gamma \triangleright e : A$

## Work sheet