Name: Hemos ID:

CSE-321 Programming Languages 2009 Final

	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 λ -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 λ -calculus:

```
\begin{array}{lll} \text{type} & A & ::= & \text{int} \mid \text{bool} \mid A \mathop{\rightarrow} A \mid A \times A \\ \text{expression} & e & ::= & x \mid \lambda x \colon 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 \colon A.\ e \\ & 0 \mid 1 \mid 2 \mid \cdots \end{array}
```

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

$(\lambda fg:$		·•		
(fix fg :				
				•
				-
)

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:

$$iarray = ref(int \rightarrow int)$$

We need the following constructs for arrays:

- new: unit→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 construct for mutable references to implement new, access and update. Fill in the blank:

new	=	λ_{-} : unit. ref λi : int. 0
access	=	λa : iarray. λi : int
ıpdate	=	λa : iarray. λi : int. λn : int.
		let

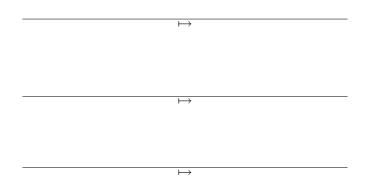
3 Evaluation context and environment [30 pts]

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

$$\begin{array}{lll} \text{type} & A & ::= & \text{unit} \mid A \rightarrow A \\ \text{expression} & e & ::= & x \mid \lambda x \colon\! A.\, e \mid e \; e \mid () \\ \text{value} & v & ::= & \lambda x \colon\! A.\, e \mid () \end{array}$$

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 $e \mapsto e'$.



Question 2. [5 pts] Assuming that the reduction relation \mapsto_{β} denotes the β -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:

$$\frac{e \mapsto_{\beta} e'}{\kappa \llbracket e \rrbracket \mapsto \kappa \llbracket e' \rrbracket} \ Red_{\beta}$$

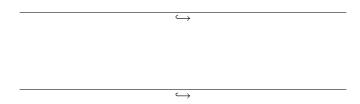
Give the definition of evaluation context which leads to the CBV strategy.

evaluation context $\kappa ::=$ _____

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

$$e \hookrightarrow v \qquad \Leftrightarrow \quad e \ evaluates \ to \ v$$

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.



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:

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

· 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.

$$\frac{\eta \vdash x \hookrightarrow}{\eta \vdash \lambda x : A. e \hookrightarrow}$$

$$\frac{\eta \vdash e_1 e_2 \hookrightarrow}{}$$

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

- $\sigma \triangleright e @ \eta$ means that the machine is currently analyzing e under the environment η . In order to evaluate a variable in e, we look up the environment η .
- $\sigma \triangleleft v$ means that the machine is currently returning v to the stack σ . 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 E. Complete the definition.

	value	v ::= $_$				
enviror	nment	$ \eta ::= \cdot \mid \eta, x \hookrightarrow v $				
	frame	ϕ ::=				
	stack	$\sigma ::= \Box \mid \sigma; \phi$				
	state	$s ::= \sigma \triangleright e @ \eta \mid \sigma \blacktriangleleft v$				
		m () N C m				
$\frac{x \hookrightarrow v \in \eta}{\sigma \blacktriangleright x @ \eta \mapsto_{E} \sigma \blacktriangleleft v}$						
-						
		⊢ E				
-		⊢→E				
-		—————————————————————————————————————				

4 Subtyping [25 pts]

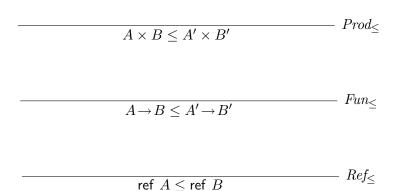
Question 1. [10 pts] The principle of subtyping is a principle specifying when a type is a subtype of another type. It states that A is a subtype of B if an expression of type A may be used whenever an expression of type B is expected. Formally we write $A \leq B$ if A is a subtype of B. The principle of subtyping justifies two subtyping rules: reflexivity and transitivity. Give the two subtyping rules:



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:



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



5 Recursive types [30 pts]

Consider the following simply-typed λ -calculus extended with recursive types:

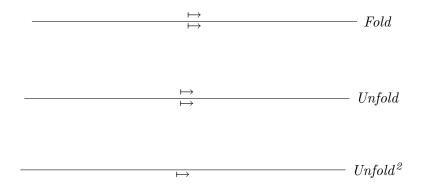
$$\begin{array}{lll} \text{type} & A & ::= & A \to A \mid \alpha \mid \mu\alpha.A \\ \text{expression} & e & ::= & x \mid \lambda x \colon\! A.\,e \mid e \mid e \mid \mathsf{fold}_C \mid e \mid \mathsf{unfold}_C \mid e \\ \text{typing context} & \Gamma & ::= & \cdot \mid \Gamma, x \colon\! A \mid \Gamma, \alpha \mid \mathsf{type} \end{array}$$

Question 1. [5 pts] Give typing rules for fold_C e and unfold_C e.

$$\frac{C = \mu \alpha.A}{\Gamma \vdash \mathsf{fold}_C \; e:} \;\; \mathsf{Fold}$$

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

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 $fold_C e$ and $unfold_C e$.



Question 3. [10 pts] Translate a recursive data type for lists of natural numbers into the simply-typed λ -calculus extended with recursive types.

datatype nlist	=	${\tt Nil} \mid {\tt Cons} \ {\tt of} \ {\tt nat} \times {\tt nlist}$
nlist	=	$\mu\alpha$
Nil	=	
$\mathtt{Cons}\ e$	=	
case e of Nil $\Rightarrow e_1 \mid \mathtt{Cons} \ x \Rightarrow e_2$	=	case of inl e_1 inr $x.e_2$

Question 4. [10 pts] 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 $\Omega \to \Omega$ 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 $\Omega \to \Omega$ because e_2 is assigned type Ω . Thus Ω must be identified with $\Omega \to \Omega$. Use recursive types and their constructs to complete the definition of Ω and e° . Fill in the blank:

$$\Omega = \underline{\qquad}$$

$$x^{\circ} = \underline{\qquad}$$

$$(\lambda x. e)^{\circ} = \underline{\qquad}$$

$$(e_1 e_2)^{\circ} = \underline{\qquad}$$

6 Polymorphism (15 pts)

There are three type systems for polymorphism: System F, Predicative Polymorphic λ -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
$$A::=$$

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

Question 2. [5 pts] The following shows the definition of Predicative Polymorphic λ -calculus. Complete the definition:

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

7 Type reconstruction [40 pts]

In this problem, we will examine the type reconstruction algorithm, called W, for the following expressions:

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

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:

$$\operatorname{id} \cdot U = U$$

$$\{A/\alpha\} \cdot \alpha = \underline{\hspace{1cm}} \text{ where } \alpha \neq \beta$$

$$\{A/\alpha\} \cdot B_1 \to B_2 = \underline{\hspace{1cm}} \text{ where } \alpha \neq \beta$$

$$\{A/\alpha\} \cdot \forall \alpha . U = \underline{\hspace{1cm}} \text{ where } \alpha \neq \beta$$

$$\{A/\alpha\} \cdot \forall \beta . U = \underline{\hspace{1cm}} \text{ where } \alpha \neq \beta$$

$$S_1 \circ S_2 \cdot U = S_1 \cdot (S_2 \cdot U)$$

Question 2. [5 pts] 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, Unify(E) returns fail. The following shows the definition of Unify(E). Complete the definition. You may use an auxiliary function ftv(A), which denotes the set of free type variables in A:

$$\mathsf{Unify}(\cdot) = \mathsf{id}$$

$$\mathsf{Unify}(E,\alpha=A) = \mathsf{Unify}(E,A=\alpha) = \mathsf{if} \underline{\hspace{1cm}}$$

$$\mathsf{else} \; \mathsf{if} \underline{\hspace{1cm}}$$

$$\mathsf{else} \; \underline{\hspace{1cm}}$$

$$\mathsf{unify}(E,A_1 \to A_2 = B_1 \to B_2) = \mathsf{Unify}(E,A_1 = B_1,A_2 = B_2)$$

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

$$\begin{aligned} \mathsf{Gen.}(\alpha \!\to\! \alpha) &=& \forall \alpha.\alpha \!\to\! \alpha \\ \\ \mathsf{Gen}_{x:\alpha}(\alpha \!\to\! \alpha) &=& \underline{\hspace{2cm}} \\ \\ \mathsf{Gen}_{x:\alpha}(\alpha \!\to\! \beta) &=& \underline{\hspace{2cm}} \\ \\ \mathsf{Gen}_{x:\alpha,y:\beta}(\alpha \!\to\! \beta) &=& \underline{\hspace{2cm}} \end{aligned}$$

Question 4. [15 pts] The type reconstruction algorithm, called 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. The following specifies the algorithm W. Complete the specification. You may use auxiliary functions $\mathsf{Unify}(E)$ and $\mathsf{Gen}_{\Gamma}(A)$:

$$\mathcal{W}(\Gamma,x) = (\mathrm{id},\{\vec{\beta}/\vec{\alpha}\} \cdot A) \qquad x: \forall \vec{\alpha}.A \in \Gamma \text{ and fresh } \vec{\beta}$$

$$\mathcal{W}(\Gamma,\lambda x.e) = \mathrm{let} (S,A) = \mathcal{W}(\Gamma+x:\alpha,e) \text{ in } \text{ fresh } \alpha$$

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

$$\mathcal{W}(\Gamma,e_1\ e_2) = \mathrm{let}$$

$$\mathrm{let}$$

$$\mathrm{let}$$

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

$$\mathrm{let}$$

$$\mathrm{l$$

Question 5. [10 pts] Until now, we consider the algorithm W for the following expressions:

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

The definitions of monotype A, polytype U and typing context Γ 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 W). If $W(\Gamma, e) = (S, A)$,		
41		

Work sheet