Name: Hemos ID:

CSE-321 Programming Languages 2006
Final — Sample Solution

	Problem 1	Problem 2	Problem 3	Problem 4	Problem 5	Problem 6	Total
Score							
Max	20	E.C.	20	20	20	20	100

1 TRUST ME – IT'S FUN! [20 pts]

I am wearing my lovely $\frac{\mathcal{PL}\ 2006}{\mathcal{I}\ \mathcal{SURVIVED}}\ \lambda$ T-shirt — Yes?

2 Good Boy, Good Girl [Extracredit ?? pts]

I attended all the lectures in this course, without missing a single lecture – Yes?

3 Generalizing a lemma [20 pts]

Throughout this problem, we use the following convention:

- SML programs are written in `typewriter` font.
  E.g., `1`, `1 + 1`, `1 * 1`, `fun`.
- Mathematical expressions and metavariables are written in *italic* or roman font.
  E.g., 1, $1+1$, n, $n+1$, $n \times 1$.
- $\widehat{n}$ stands for an SML constant expression for a mathematical integer n.
- $e \mapsto e'$ stands for a reduction of SML program e to SML program e' under the usual reduction strategy for SML. $e \mapsto^* e'$ stands for its reflexive and transitive closure.

Examples:

$$
\begin{array}{rcl}
\widehat{0} & = & \texttt{0} \\
\widehat{1} & = & \texttt{1} \\
\widehat{n+1}\ \texttt{-}\ \texttt{1} & \neq & \widehat{n} \\
\widehat{n+1}\ \texttt{-}\ \texttt{1} & \mapsto & \widehat{n} \\
\widehat{n}\ \texttt{*}\ \texttt{1} & \mapsto & \widehat{n} \\
\widehat{n}\ \texttt{*}\ \widehat{m} & \mapsto & \widehat{n \times m} \\
\widehat{n}\ \texttt{*}\ (\widehat{m}\ \texttt{*}\ \widehat{l}) & \mapsto^* & \widehat{n \times m \times l}
\end{array}
$$

Consider the following SML tail-recursive implementation of the factorial function:

```
fun fact' 0 a = a
  | fact' n a = fact' (n - 1) (n * a)

fun fact n = fact' n 1
```

Our goal is to prove that `fact` $\widehat{n}$ evaluates to $\widehat{n!}$ by mathematical induction on n:

Theorem 1. `fact` $\widehat{n} \mapsto^* \widehat{n!}$ holds for any natural number $n \geq 0$.

Since `fact` $\widehat{n}$ reduces to `fact'` $\widehat{n}$ `1`, we try to prove the following lemma instead:

Lemma 2. `fact'` $\widehat{n}$ `1` $\mapsto^* \widehat{n!}$ holds for any natural number $n \geq 0$.

Lemma 2 is impossible to prove by mathematical induction on n, however. Here is a failed attempt to prove Lemma 2:

Failed proof of Lemma 2. By mathematical induction on n.

Base case $n = 0$.

$$\begin{array}{rl} & \texttt{fact'}\ \widehat{0}\ \texttt{1} \\ \mapsto & \texttt{1} \\ = & \widehat{0!} \end{array}$$

Inductive case $n > 0$.

Induction hypothesis: assume $\texttt{fact'}\ \widehat{n}\ \texttt{1} \mapsto^* \widehat{n!}$.

Induction step:

$$\begin{array}{rl} & \texttt{fact'}\ \widehat{n+1}\ \texttt{1} \\ \mapsto & \texttt{fact'}\ (\widehat{n+1}\ \texttt{-}\ \texttt{1})\ (\widehat{n+1}\ \texttt{*}\ \texttt{1}) \\ \mapsto & \texttt{fact'}\ \widehat{n}\ (\widehat{n+1}\ \texttt{*}\ \texttt{1}) \\ \mapsto & \texttt{fact'}\ \widehat{n}\ \widehat{n+1} \end{array}$$

Now we cannot proceed further because the second argument $\widehat{n+1}$ to `fact'` cannot be `1` and the induction hypothesis is not applicable. □

Your task is to find a lemma on `fact'` that generalizes Lemma 2 in the sense that it directly implies Lemma 2. For example, $\widehat{m}\ \texttt{*}\ \widehat{n} \mapsto \widehat{m \times n}$ is a generalization of $\widehat{m}\ \texttt{*}\ \widehat{1} \mapsto \widehat{m \times 1}$ because the second is just an instance of the first. Your lemma should be designed in such a way that Theorem 1 follows from it.

Instruction:

1. Fill in the blank below.
2. Be careful about distinguishing between SML programs and mathematical expressions. In particular, use `*` for multiplication in SML and $\times$ for mathematical multiplication.
3. In your proof, all mathematical expressions must be under the hat symbol $\widehat{\ }$.
4. Organize your proof in accordance with symbols $\mapsto$ and $=$ provided.
5. Grading for this problem will be **strict**.
6. Hint: There is no general recipe for generalizing a lemma, but for this problem, identifying a proper invariant of `fact'` will do the job. (Invariants are such an important tool!)

Lemma 3. `fact'` $\widehat{n}\ \widehat{a} \mapsto^* \widehat{n! \times a}$ holds for any natural number $n \geq 0$ and a.

Proof. By mathematical induction on n.

Base case $n = 0$.

`fact'` $\widehat{0}\ \widehat{a}$

$\mapsto$ $\widehat{a}$

$=$ $\widehat{0! \times a}$

Inductive case $n > 0$.

Induction hypothesis: assume `fact'` $\widehat{n}\ \widehat{a} \mapsto^* \widehat{n! \times a}$.

Induction step:

`fact'` $\widehat{n+1}\ \widehat{a}$

$\mapsto$ `fact' (`$\widehat{n+1}$ `- 1) (`$\widehat{n+1}$ `*` $\widehat{a}$`)`

$\mapsto$ `fact'` $\widehat{n}$ `(`$\widehat{n+1}$ `*` $\widehat{a}$`)`

$\mapsto$ `fact'` $\widehat{n}$ `(`$\widehat{(n+1) \times a}$`)`

$\mapsto^*$ $n! \times (\widehat{(n+1) \times a})$ by induction hypothesis

$=$ $\widehat{(n+1)! \times a}$

□

Now we are ready to prove Theorem 1 using Lemma 3:

Proof of Theorem 1.

`fact` $\widehat{n}$

$\mapsto$ `fact'` $\widehat{n}$ `1`

$\mapsto^*$ $\widehat{n! \times 1}$ by Lemma 3

$=$ $\widehat{n!}$

□

4 Exceptions [20 pts]

Consider the abstract machine C for the simply typed λ-calculus:

$$\begin{array}{rrcl}
\textsf{type} & A & ::= & P \mid A \to A \\
\textsf{base type} & P & & \\
\textsf{expression} & e & ::= & x \mid \lambda x\!:\!A.\, e \mid e\ e \\
\textsf{value} & v & ::= & \lambda x\!:\!A.\, e \\
\textsf{frame} & \phi & ::= & \Box\ e \mid v\ \Box \\
\textsf{stack} & \sigma & ::= & \Box \mid \sigma; \phi \\
\textsf{state} & s & ::= & \sigma \blacktriangleright e \mid \sigma \blacktriangleleft v
\end{array}$$

The goal of this problem is to extend the abstract machine C with *exceptions*.

To begin with, we introduce two new forms of expressions $\mathsf{try}\ e\ \mathsf{with}\ e'$ and exn:

$$\textsf{expression} \qquad e \ ::= \ \cdots \mid \mathsf{try}\ e\ \mathsf{with}\ e \mid \mathsf{exn}$$

Informally $\mathsf{try}\ e\ \mathsf{with}\ e'$ tries to evaluate e. If e successfully evaluates to v, then $\mathsf{try}\ e\ \mathsf{with}\ e'$ also evaluates to the same value v. In this case, e' is never visited and is thus ignored. If the evaluation of e raises an exception by attempting to reduce exn, then the evaluation of e is canceled and e' is evaluated instead. In this way, $\mathsf{try}\ e\ \mathsf{with}\ e'$ handles every exception raised within e. Note that e' itself may also raise an exception, in which case the exception propagates to the next $\mathsf{try}\ e_{next}\ \mathsf{with}\ e'_{next}$ such that e_{next} encloses $\mathsf{try}\ e\ \mathsf{with}\ e'$.

Formally we extend the operational semantics with the following reduction rules.

$$\frac{}{\mathsf{exn}\ e \mapsto \mathsf{exn}}\ \mathit{Exn} \qquad \frac{}{v\ \mathsf{exn} \mapsto \mathsf{exn}}\ \mathit{Exn}'$$

$$\frac{e_1 \mapsto e_1'}{\mathsf{try}\ e_1\ \mathsf{with}\ e_2 \mapsto \mathsf{try}\ e_1'\ \mathsf{with}\ e_2}\ \mathit{Try} \qquad \frac{}{\mathsf{try}\ v\ \mathsf{with}\ e \mapsto v}\ \mathit{Try}' \qquad \frac{}{\mathsf{try}\ \mathsf{exn}\ \mathsf{with}\ e \mapsto e}\ \mathit{Try}''$$

Give new rules for the reduction judgment $s \mapsto_{\mathsf{C}} s'$ corresponding to the above five reduction rules. You have to use an additional state $\sigma \blacktriangleleft\blacktriangleleft \mathsf{exn}$:

$$\textsf{state} \qquad s \ ::= \ \cdots \mid \sigma \blacktriangleleft\blacktriangleleft \mathsf{exn}$$

- A state $\sigma \blacktriangleleft\blacktriangleleft \mathsf{exn}$ means that the machine is currently propagating an exception exn.

Instruction:

1. You will need exactly five rules.
2. Write only those rules related with $\mathsf{try}\ e\ \mathsf{with}\ e'$ and exn. For example, do not copy the rules from Course Notes.

1.

$$\frac{}{\sigma \blacktriangleright \mathsf{try}\ e_1\ \mathsf{with}\ e_2 \mapsto_{\mathsf{C}} \sigma; \mathsf{try}\ \square\ \mathsf{with}\ e_2 \blacktriangleright e_1}$$

2.

$$\frac{}{\sigma; \mathsf{try}\ \square\ \mathsf{with}\ e_2 \blacktriangleleft v \mapsto_{\mathsf{C}} \sigma \blacktriangleleft v}$$

3.

$$\frac{}{\sigma \blacktriangleright \mathsf{exn} \mapsto_{\mathsf{C}} \sigma \blacktriangleleft\blacktriangleleft \mathsf{exn}}$$

4.

$$\frac{\phi \neq \mathsf{try}\ \square\ \mathsf{with}\ e}{\sigma; \phi \blacktriangleleft\blacktriangleleft \mathsf{exn} \mapsto_{\mathsf{C}} \sigma \blacktriangleleft\blacktriangleleft \mathsf{exn}}$$

5.

$$\frac{}{\sigma; \mathsf{try}\ \square\ \mathsf{with}\ e \blacktriangleleft\blacktriangleleft \mathsf{exn} \mapsto_{\mathsf{C}} \sigma \blacktriangleright e}$$

5 Simulating call-by-name with call-by-value [20 pts]

An evaluation strategy (or reduction strategy) is a set of rules that specify how to evaluate expressions using the β-reduction. In Chapter 3 of Course Notes, we have learned two reduction strategies: *call-by-name* (CBN) and *call-by-value* (CBV).

The goal of this problem is to simulate CBN with CBV. (The problem of simulating CBV with CBN is a lot more involved.) We first define the abstract syntax for CBN and CBV simply typed λ-calculi. To distinguish CBN function types from CBV function types, we use $A \supset B$ for functions from type A to type B under CBN; as usual, we use $A \to B$ for functions from type A to type B under CBV.

The abstract syntax for the CBN λ-calculus as follows; we include $\widehat{n}$ for integers of type int:

$$\begin{array}{rrcl} \textsf{type} & A & ::= & \mathsf{int} \mid A \supset A \\ \textsf{expression} & e & ::= & x \mid \lambda x{:}A.\, e \mid e\ e \mid \widehat{n} \end{array}$$

The abstract syntax for the CBV λ-calculus includes a unit value () of type unit:

$$\begin{array}{rrcl} \textsf{type} & A & ::= & \mathsf{int} \mid A \to A \mid \mathsf{unit} \\ \textsf{expression} & e & ::= & x \mid \lambda x{:}A.\, e \mid e\ e \mid \widehat{n} \mid () \end{array}$$

To simulate CBN with CBV, we introduce two translation functions: A^* for types and $\overline{e}$ for expressions. A^* is a type in the CBV λ-calculus corresponding to type A in the CBN λ-calculus. That is, A^* is a CBV type obtained by translating a CBN type A. Likewise $\overline{e}$ is a CBV expression obtained by translating a CBN expression e. The translation of a typing context Γ in CBN is also represented by Γ^*. A^* and Γ^* are inductively defined as follows:

$$\begin{array}{rcl} \mathsf{int}^* & = & \mathsf{int} \\ (A \supset B)^* & = & (\mathsf{unit} \to A^*) \to B^* \\ (\Gamma, x : A)^* & = & \Gamma^*, x : \mathsf{unit} \to A^* \end{array}$$

Since a function argument under the CBN strategy is not evaluated until it is actually used, we translate a type A for function arguments in the CBN λ-calculus into $\mathsf{unit} \to A^*$ in the CBV λ-calculus. The idea is that in the CBV λ-calculus, we can delay the evaluation of an expression e by enclosing it within a λ-abstraction $\lambda \_{:}\mathsf{unit}.\, e$. Finally each variable x itself in a typing context Γ is not changed by the translation, but its associated type A is translated into $\mathsf{unit} \to A^*$.

Question 1. **[10 pts]** Complete the definition of the translation function $\overline{e}$ for expressions. The translation function $\overline{e}$ should be designed so that it satisfies the following invariant:

Invariant. *If* $\Gamma \vdash e : A$ *holds in the CBN* λ*-calculus, then* $\Gamma^* \vdash \overline{e} : A^*$ *holds in the CBV* λ*-calculus.*

Fill in the blank, keeping in mind the definition of A^* and the invariant given above.

$$\begin{aligned}
\overline{x} &= x\ () \\
\overline{\lambda x\!:\!A.\, e} &= \lambda x\!:\!\mathsf{unit} \to A^*.\, \overline{e} \\
\overline{e_1\ e_2} &= \overline{e_1}\ (\lambda_\!:\!\mathsf{unit}.\, \overline{e_2}) \\
\overline{\widehat{n}} &= \widehat{n}
\end{aligned}$$

Question 2. **[10 pts]** Using the above translation function, translate the following CBN expression into a CBV expression and then show its reduction sequence *under the CBV strategy.*

$$\begin{aligned}
& \overline{(\lambda x\!:\!\mathsf{int}.\, x)\ ((\lambda x\!:\!\mathsf{int}.\, x)\ \widehat{0})} \\
= \quad & (\lambda x\!:\!\mathsf{unit} \to \mathsf{int}.\, x\ ())\ (\lambda_\!:\!\mathsf{unit}.\, ((\lambda x\!:\!\mathsf{unit} \to \mathsf{int}.\, x\ ())\ (\lambda_\!:\!\mathsf{unit}.\, \widehat{0}))) \\
\mapsto_{CBV} \quad & (\lambda_\!:\!\mathsf{unit}.\, ((\lambda x\!:\!\mathsf{unit} \to \mathsf{int}.\, x\ ())\ (\lambda_\!:\!\mathsf{unit}.\, \widehat{0})))\ () \\
\mapsto_{CBV} \quad & (\lambda x\!:\!\mathsf{unit} \to \mathsf{int}.\, x\ ())\ (\lambda_\!:\!\mathsf{unit}.\, \widehat{0}) \\
\mapsto_{CBV} \quad & (\lambda_\!:\!\mathsf{unit}.\, \widehat{0})\ () \\
\mapsto_{CBV} \quad & \widehat{0}
\end{aligned}$$

6 Recursion with recursive types [20 pts]

Consider the (call-by-value) simply typed λ-calculus with recursive types, which we refer to as $\mathcal{L}_1$. For the sake of simplicity, we do not annotate fold and unfold with a recursive type:

$$\mathcal{L}_1 \;=\; \left\{ \begin{array}{rrcl} \mathsf{type} & A & ::= & A \to A \mid \alpha \mid \mu\alpha.A \\ \mathsf{expression} & e & ::= & x \mid \lambda x{:}A.\, e \mid e\ e \mid \mathsf{fold}\ e \mid \mathsf{unfold}\ e \\ \mathsf{value} & v & ::= & \lambda x{:}A.\, e \mid \mathsf{fold}\ v \end{array} \right.$$

We have seen in Chapter 13 of Course Notes that $\mathcal{L}_1$ permits recursive functions even without the fixed point construct. The idea is that we translate the fixed point combinator fix of the untyped λ-calculus to fix° of $\mathcal{L}_1$. Unfortunately fix° is not particularly useful because it has type $\Omega = \mu\alpha.\alpha \to \alpha$. For example, we cannot use fix° to directly implement a recursive function on integers (such as factorial function, Fibonacci function, and so on) as we would in SML.

In this problem, we will investigate how to directly implement recursive functions in $\mathcal{L}_1$ using recursive types. We first extend $\mathcal{L}_1$ with a new form of type $A \Rightarrow B$ and two new forms of expressions $\mathsf{fun}\ f\ x{:}A.\, e$ and $e \odot e$; we refer to the resultant language as $\mathcal{L}_2$:

$$\mathcal{L}_2 \;=\; \mathcal{L}_1 + \left\{ \begin{array}{rrcl} \mathsf{type} & A & ::= & \cdots \mid A \Rightarrow A \\ \mathsf{expression} & e & ::= & \cdots \mid \mathsf{fun}\ f\ x{:}A.\, e \mid e \odot e \\ \mathsf{value} & v & ::= & \cdots \mid \mathsf{fun}\ f\ x{:}A.\, e \end{array} \right.$$

A *recursive function construct* $\mathsf{fun}\ f\ x{:}A.\, e$ defines a recursive function f whose argument is x of type A and whose body is e. A *recursive function application* $e_1 \odot e_2$ applies a recursive function obtained by evaluating e_1 to the result of evaluating e_2. We use a *recursive function type* $A \Rightarrow B$ for recursive functions from type A to type B.

$$\frac{\Gamma, f : A \Rightarrow B, x : A \vdash e : B}{\Gamma \vdash \mathsf{fun}\ f\ x{:}A.\, e : A \Rightarrow B}\ \Rightarrow\!\mathsf{I} \qquad \frac{\Gamma \vdash e : A \Rightarrow B \quad \Gamma \vdash e' : A}{\Gamma \vdash e \odot e' : B}\ \Rightarrow\!\mathsf{E}$$

$$\frac{e_1 \mapsto e_1'}{e_1 \odot e_2 \mapsto e_1' \odot e_2}\ \mathit{Fun} \qquad \frac{e_2 \mapsto e_2'}{(\mathsf{fun}\ f\ x{:}A.\, e) \odot e_2 \mapsto (\mathsf{fun}\ f\ x{:}A.\, e) \odot e_2'}\ \mathit{RArg}$$

$$\frac{}{(\mathsf{fun}\ f\ x{:}A.\, e) \odot v \mapsto [\mathsf{fun}\ f\ x{:}A.\, e/f][v/x]e}\ \mathit{RApp}$$

The goal is to show that $\mathcal{L}_2$ is equivalent to $\mathcal{L}_1$: adding $\mathsf{fun}\ f\ x{:}A.\, e$ and $e \odot e$ to $\mathcal{L}_1$ does not actually increase its expressive power. To be precise, you will show that:

- $A \Rightarrow B$ can be defined in terms of ordinary function types and recursive types in $\mathcal{L}_1$.
- $\mathsf{fun}\ f\ x{:}A.\, e$ and $e \odot e$ can be defined in terms of those constructs in $\mathcal{L}_1$.

Question 1. **[10 pts]** Complete the definition of a translation function $(\cdot)^\star$ which translates a type A in $\mathcal{L}_2$ to a type $A^\star$ in $\mathcal{L}_1$.

Hint. The key idea in the translation is essentially the same as in developing the fixed point combinator fix for the untyped λ-calculus: *self-application*. Informally speaking, a recursive function takes itself as its first argument. Perhaps the rules $\Rightarrow\mathsf{I}$ and *RApp* are the best hint that we could give.

Fill in the blank:

$$\begin{array}{rcl}
(A \to B)^\star & = & A^\star \to B^\star \\
\alpha^\star & = & \alpha \\
(\mu\alpha.A)^\star & = & \mu\alpha.A^\star \\
\\
(A \Rightarrow B)^\star & = & \underline{\quad \mu\alpha.\alpha \to (A^\star \to B^\star) \quad}
\end{array}$$

Question 2. **[5 pts]** Complete the definition of a translation function $(\cdot)^\bullet$ which translates an expression e in $\mathcal{L}_2$ to an expression $e^\bullet$ in $\mathcal{L}_1$. The translation function must satisfy the following invariant:

Invariant. *If* $\Gamma \vdash e : A$ *holds in* $\mathcal{L}_2$*, then* $\Gamma^\star \vdash e^\bullet : A^\star$ *holds in* $\mathcal{L}_1$*, where* $\Gamma^\star$ *converts each type binding* $x : A$ *in* Γ *to* $x : A^\star$*.*

Fill in the blank below. For the case $(\mathsf{fun}\ f\ x{:}A.\, e)^\bullet$, we assume that $\mathsf{fun}\ f\ x{:}A.\, e$ has type $A \Rightarrow B$ in $\mathcal{L}_2$.

$$\begin{array}{rcl}
x^\bullet & = & \mathrm{x} \\
(\lambda x{:}A.\, e)^\bullet & = & \lambda x{:}A^\star.\, e^\bullet \\
(e_1\ e_2)^\bullet & = & e_1{}^\bullet\ e_2{}^\bullet \\
(\mathsf{fold}\ e)^\bullet & = & \mathsf{fold}\ e^\bullet \\
(\mathsf{unfold}\ e)^\bullet & = & \mathsf{unfold}\ e^\bullet \\
\\
(\mathsf{fun}\ f\ x{:}A.\, e)^\bullet & = & \underline{\quad \mathsf{fold}\ \lambda f{:}(A \Rightarrow B)^\star.\, \lambda x{:}A^\star.\, e^\bullet \quad} \\
\\
(e_1 \odot e_2)^\bullet & = & \underline{\quad \mathsf{unfold}\ e_1{}^\bullet\ e_1{}^\bullet\ e_2{}^\bullet \quad}
\end{array}$$

Question 3. **[5 pts]** Show the reduction sequence of $((\mathsf{fun}\ f\ x{:}A.\, e) \odot v)^\bullet$. We assume that $\mathsf{fun}\ f\ x{:}A.\, e$ has type $A \Rightarrow B$ in $\mathcal{L}_2$. We also assume that no variable capture occurs in substitutions during the reduction.

$$\begin{array}{rcl}
 & & ((\mathsf{fun}\ f\ x{:}A.\, e) \odot v)^\bullet \\
 & = & \underline{\quad \mathsf{unfold}\ (\mathsf{fold}\ \lambda f{:}(A \Rightarrow B)^\star.\, \lambda x{:}A^\star.\, e^\bullet)\ (\mathsf{fun}\ f\ x{:}A.\, e)^\bullet\ v^\bullet \quad} \\
 & \mapsto & \underline{\quad (\lambda f{:}(A \Rightarrow B)^\star.\, \lambda x{:}A^\star.\, e^\bullet)\ (\mathsf{fun}\ f\ x{:}A.\, e)^\bullet\ v^\bullet \quad} \\
 & \mapsto & \underline{\quad (\lambda x{:}A^\star.\, [(\mathsf{fun}\ f\ x{:}A.\, e)^\bullet / f] e^\bullet)\ \ v^\bullet \quad} \\
 & \mapsto & \underline{\quad [v^\bullet / x][(\mathsf{fun}\ f\ x{:}A.\, e)^\bullet / f] e^\bullet \quad}
\end{array}$$