Fall 1998 CS611 Final

DUE : 5pm Friday, 4 Dec 1998

This is your final exam for CS611. The exam is open book and open notes. However, you should refrain from talking with anyone (other than Stephanie or Greg) about this exam, and you should not go looking in other text books or sources for answers to the questions. Please do not post questions directly to the class newsgroup. Instead, please contact either Stephanie (`sweirich@cs.cornell.edu`) or Greg (`jgm@cs.cornell.edu`). If I even suspect that you've cheated, you will be kicked out of the program.

You have one work week to finish the exam. Please turn in all of your materials to Joan Lockwood (Upson 4147) or Greg's office (Upson 4133) by no later than 5pm on Friday, December 4. Make sure that your solutions are neat – we will count off for poor handwriting.

If in doubt regarding a question, or if you find a bug, please contact us sooner rather than later. Read the class bulletin board and the class home page, as we will post fixes and/or clarifications there.

1. **Operational Semantics:** We wish to add a `break` command to IMP, similar to the one provided by languages like C or Java. The intended semantics is that, if we execute `break`, then we exit out of nearest enclosing `while` loop (if any). If there is no enclosing while loop, then we immediately terminate the program.

   Give a new small-step operational semantics for IMP + `break`. You should define carefully your configurations and your one-step rewriting rules for commands. (Hint: you'll need to add something to configurations.)

2. **The Simply-Typed Lambda Calculus with Lists:** As discussed in class, every program in the simply-typed lambda calculus ($\lambda^{\rightarrow}$) ter-

minates. On the other hand, the simply-typed lambda calculus is, well, rather simple.

In this question, you will study a slightly more complex language called λ^{list} which, instead of providing integers, provides only unit as the base type. However, λ^{list} also provides lists and a term `fold` which can be used to manipulate lists.

The syntax of λ^{list} is as follows:

$$
\begin{array}{llll}
\text{(types)} & \tau & ::= & \mathsf{unit} \mid \tau_1 \to \tau_2 \mid \mathsf{list}[\tau] \\
\text{(terms)} & e & ::= & \langle\,\rangle \mid x \mid \lambda x{:}\tau.e \mid e_1\, e_2 \mid \mathtt{nil}_\tau \\
 & & & \mid \mathtt{cons}(e_1, e_2) \mid \mathtt{fold}(e_1, e_2, e_3) \\
\text{(values)} & v & ::= & \langle\,\rangle \mid \lambda x{:}\tau.e \mid \mathtt{nil}_\tau \mid \mathtt{cons}(v_1, v_2)
\end{array}
$$

The static semantics is the same as for $\lambda^{\to}$ except that we replace the rule for integers with one for unit:

$$\Gamma \vdash \langle\,\rangle : \mathsf{unit}$$

and add the following rules for lists:

$$\Gamma \vdash \mathtt{nil}_\tau : \mathsf{list}[\tau]$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \mathsf{list}[\tau]}{\Gamma \vdash \mathtt{cons}(e_1, e_2) : \mathsf{list}[\tau]}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_2 \qquad \Gamma \vdash e_3 : \mathsf{list}[\tau_1]}{\Gamma \vdash \mathtt{fold}(e_1, e_2, e_3) : \tau_2}$$

The small-step operational semantics is the same as for the call-by-name version of $\lambda^{\to}$ but with the addition of the following rules:

$$\frac{e_1 \mapsto e_1'}{\mathtt{cons}(e_1, e_2) \mapsto \mathtt{cons}(e_1', e_2)}$$

$$\frac{e_2 \mapsto e_2'}{\mathtt{cons}(v, e_2) \mapsto \mathtt{cons}(v, e_2')}$$

$$\frac{e_3 \mapsto e_3'}{\mathtt{fold}(e_1, e_2, e_3) \mapsto \mathtt{fold}(e_1, e_2, e_3')}$$

$$\mathtt{fold}(e_1, e_2, \mathtt{nil}_\tau) \mapsto e_2$$

$$\mathtt{fold}(e_1, e_2, \mathtt{cons}(v_3, v_4)) \mapsto \mathtt{fold}(e_1, (e_1\, v_3\, e_2), v_4)$$

It turns out that, as we shall see, `fold` is an extremely useful operation.

(a) Show how you can encode natural numbers within λ^{list}. That is, give a value term (v_0) that you will use to represent 0 (zero) and, given a value v_n that represents the number n, show how to construct a value v_{n+1} that represents $n+1$.

(b) Write a λ^{list} function v_+ that takes the representation of two numbers n and m and returns the representation of $n+m$. Prove that for all v_n and v_m, the compound term $v_+\, v_n\, v_m$ evaluates to v_{n+m}.

(c) Show how you can encode booleans within λ^{list}. That is, give two terms v_t and v_f which represent true and false respectively. Then show how you can represent "`if` e_1 `then` e_2 `else` e_3" as a term within λ^{list}. In particular, if e_1 evaluates to v_t, then the above expression should evaluate to whatever e_2 evaluates to, and if e_1 evaluates to v_f, then the above expression should evaluate to whatever e_3 evaluates to.

(d) The `fold` operation must evaluate its first agrgument for each list element that occurs in its third argument. Suggest an alternative set of rules for `fold` that behaves the same, but avoids re-computing the first argument.

(e) (*Hard:*) At this point, it is fairly clear that you can encode booleans and some arithmetic within λ^{list} and so we may conjecture that the language is Turing complete. However, this is not the case. Prove that every term in the language terminates by extending the proof (given in class) that every term in $\lambda^{\rightarrow}$ terminates, thereby establishing that the halting problem for this language is trivially decidable.

3. **Call-by-Name Denotational Semantics:** In class, we studied a denotational semantics for the call-by-value variant of PCF. For this exercise, you will develop a call-by-name denotational semantics and put it to work.

To remind you, the syntax of PCF types and terms is as follows:

$$
\begin{array}{llcl}
\text{(types)} & \tau & ::= & \mathsf{int} \mid \mathsf{unit} \mid \tau_1 \times \tau_2 \mid \mathsf{void} \mid \tau_1 + \tau_2 \mid \tau_1 \to \tau_2 \\
\text{(primops)} & p & ::= & + \mid - \mid \times \mid \leq \\
\text{(terms)} & e & ::= & x \mid i \mid p(e_1, e_2) \mid \langle\,\rangle \mid \langle e_1, e_2 \rangle \mid \pi_1\, e \mid \pi_2\, e \mid \mathtt{in}_1^{\tau_1+\tau_2}(e) \\
 & & & \mid \mathtt{in}_2^{\tau_1+\tau_2}(e) \mid (\mathsf{case}\, e_1\, \mathsf{of}\, \mathtt{in}_1(x_1) \texttt{=>} e_1 \mid \mathtt{in}_2(x_2) \texttt{=>} e_2) \\
 & & & \mid \lambda x{:}\tau.e \mid e_1\, e_2 \mid \mathtt{fix}(\lambda x{:}\tau.e)
\end{array}
$$

The values for the call-by-name variant of the language are as follows:

$$
\text{(values)} \quad v \quad ::= \quad i \mid \langle\,\rangle \mid \langle e_1, e_2 \rangle \mid \mathtt{in}_i^{\tau_1+\tau_2}(e) \mid \lambda x{:}\tau.e
$$

Notice that nested expressions need not be evaluated to values as in the call-by-value setting. The typing rules are the same as for the call-by-value language, but to remind you, we've included a few of the more unusual rules below.

$$
\frac{\Gamma \vdash e_1 : \mathsf{int} \quad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash \leq (e_1, e_2) : \mathsf{unit} + \mathsf{unit}}
$$

$$
\frac{\Gamma \vdash e_1 : \mathsf{int} \quad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash +(e_1, e_2) : \mathsf{int}}
$$

$$
\frac{\Gamma \vdash e : \tau_i}{\Gamma \vdash \mathtt{in}_i^{\tau_1+\tau_2}(e) : \tau_1 + \tau_2}(i = 1, 2)
$$

$$
\frac{\Gamma \vdash e : \tau_1 + \tau_2 \quad \Gamma, x_1{:}\tau_1 \vdash e_1 : \tau \quad \Gamma, x_2{:}\tau_2 \vdash e_2 : \tau}{\Gamma \vdash \mathsf{case}\, e\, \mathsf{of}\, \mathtt{in}_1(x_1) \texttt{=>} e_1 \mid \mathtt{in}_2(x_2) \texttt{=>} e_2 : \tau}
$$

$$
\frac{\Gamma \vdash \lambda x{:}\tau.e : \tau \to \tau}{\Gamma \vdash \mathtt{fix}(\lambda x{:}\tau.e) : \tau}
$$

Notice that the less-than-or-equal operation returns a "boolean" encoded as a $\mathsf{unit} + \mathsf{unit}$ value. Also notice that, unlike the call-by-value setting, we have *not* restricted the type of the bound variable for `fix` expressions.

The call-by-name, large-step operational semantics for the language is given below.

$$
v \Downarrow v \qquad \frac{e_1 \Downarrow i_1 \quad e_2 \Downarrow i_2}{+(e_1, e_2) \Downarrow i}(i_1 + i_2 = i)
$$

$$\frac{e_1 \Downarrow i_1 \quad e_2 \Downarrow i_2}{-(e_1, e_2) \Downarrow i}(i_1 - i_2 = i) \qquad \frac{e_1 \Downarrow i_1 \quad e_2 \Downarrow i_2}{*(e_1, e_2) \Downarrow i}(i_1 * i_2 = i)$$

$$\frac{e_1 \Downarrow i_1 \quad e_2 \Downarrow i_2}{\leq (e_1, e_2) \Downarrow \mathtt{in}_1^{\mathsf{unit}+\mathsf{unit}}(\langle\rangle)}(i_1 \leq i_2)$$

$$\frac{e_1 \Downarrow i_1 \quad e_2 \Downarrow i_2}{\leq (e_1, e_2) \Downarrow \mathtt{in}_2^{\mathsf{unit}+\mathsf{unit}}(\langle\rangle)}(i_1 \not\leq i_2)$$

$$\frac{e \Downarrow \langle e_1, e_2\rangle \quad e_i \Downarrow v}{\pi_i\, e \Downarrow v}(i = 1, 2)$$

$$\frac{e \Downarrow \mathtt{in}_i^{\tau_1+\tau_2}(e') \quad e_i[e'/x_i] \Downarrow v}{\mathtt{case}\, e\, \mathtt{of}\ \mathtt{in}_1(x_1)\mathtt{=>}e_1 \mid \mathtt{in}_2(x_2)\mathtt{=>}e_2 \Downarrow v}(i = 1, 2)$$

$$\frac{e_1 \Downarrow \lambda x{:}\tau.e' \quad e'[e_2/x] \Downarrow v}{e_1\, e_2 \Downarrow v}$$

$$\frac{e[\mathtt{fix}(\lambda x{:}\tau.e)/x] \Downarrow v}{\mathtt{fix}(\lambda x{:}\tau.e) \Downarrow v}$$

(a) Define a sound denotational semantics for the call-by-name language. Your semantics should define:

- $\mathcal{T}[\![\tau]\!]$, a function that maps types to CPOs. That is, for each type, assign it a CPO, specifying what the underlying set is as well as the partial order that you will use on it. You need not show that the partial order is complete.
- $\mathcal{T}[\![\Gamma]\!]$, a set of environments suitable for the type-assignment Γ.
- $\mathcal{E}[\![\Gamma \vdash e : \tau]\!]$, a function from $\mathcal{T}[\![\Gamma]\!]$ to $\mathcal{T}[\![\tau]\!]$. You need not prove formally that your definitions are well-formed (e.g., that some function is continuous) but you should make it clear to us why you defined things the way you did. Be very careful in your definitions to keep track of "types" at the mathematical level. For instance, don't just write $\perp$. Rather, write $\perp_\tau$ where τ is the appropriate type, and argue that indeed such a least element exists in the CPO $\mathcal{T}[\![\tau]\!]$. When in doubt, add text to explain why you defined things the way you did.

(b) Using pencil and paper or using ML, write a compiler $\mathcal{C}$ that maps IMP commands to call-by-name PCF programs. You may assume that IMP locations (X) have associated with them a unique integer index that can be calculated via a function `locindex`. Your compiler should consist of:

i. A mapping $\mathcal{S}[\![\sigma]\!]$ from IMP states to PCF terms.

ii. A mapping $\mathcal{A}[\![a]\!]$ from IMP arithmatic expressions to PCF terms.

iii. A mapping $\mathcal{B}[\![b]\!]$ from IMP boolean expressions to PCF terms.

iv. A mapping $\mathcal{C}[\![c]\!]$ from IMP commands to PCF terms.

If you choose to use ML, then you will need to construct datatypes for the IMP and PCF languages. However, you need not write a type-checkers, evaluators, etc.

(c) Prove that your compiler has the property that if $\langle c, \sigma_0 \rangle \longrightarrow \sigma$, then $\mathcal{C}[\![c]\!]\ \mathcal{S}[\![\sigma_0]\!]\ \Downarrow \mathcal{S}[\![\sigma]\!]$. You may assume that your denotational semantics is both sound and adequate with respect to the operational semantics given. Hence, the above could be rephrased as showing that

$$\mathcal{E}[\![\vdash \mathcal{C}[\![c]\!]\ \mathcal{S}[\![\sigma_0]\!]\ : \tau_{\text{state}}]\!] = \mathcal{E}[\![\vdash \mathcal{S}[\![\sigma]\!]\ : \tau_{\text{state}}]\!]$$

where τ_{state} is the PCF type that you use to represent states.

4. What was the most interesting thing you learned in 611? What would you have liked to learn about that we didn't cover? (Note: this question, obviously, has no wrong answers. However, leaving this question blank or giving non-constructive answers will be frowned upon.)