Cornell University
Computer Science 611

Problem Set 4 DUE: 5 November 2004

Instructions

Turn in the written part of the assignment by 5PM on the due date in Upson 4119.

1. Induction (25 pts)

   Prove the following assertions using well-founded induction. Make sure to clearly identify what induction is being performed on, to state the induction hypothesis and point out where it is being used.

   (a) Given a term e in the untyped lambda calculus, show that it doesn't matter in what order you substitute closed terms. That is, given closed terms e_1, e_2, we have

   $$e\{e_1/x\}\{e_2/y\} = e\{e_2/y\}\{e_1/x\}$$

   assuming $x \neq y$.

   (b) Here's a fact we used in the soundness proof for the soundness/safety proof of the simply typed lambda calculus: the free variables of a well-typed term are always found in its typing environment:

   $$\Gamma \vdash e : \tau \Longrightarrow FV(e) \subseteq \mathrm{dom}(\Gamma)$$

2. First-class Modules (25 pts)

   The simple module values we talked about earlier in class had more expressive power than those in many programming languages in that they were *first-class values*. In fact, they look suspiciously close to objects in an object-oriented language.

   Suppose we extend call-by-value uML with the following expression forms similar to those we saw in the simpler module mechanism presented in class:

   $e ::= \ldots \mid \mathsf{module}\ m\ (\mathsf{fields}\ X; \mathsf{methods}\ Y) \mid e.x \mid \mathsf{extend}\ e_1\ \mathsf{by}\ e_2$
   $f ::= \lambda x.\, e$
   $X ::= x_1 = e_1, \ldots, x_n = e_n$
   $Y ::= y_1 = f_1, \ldots, y_n = f_n$

   In this new language, uML$_m$, a module expression introduces a name for the module value itself, m. This identifier is in scope in the remainder of the module expression and may be used in the method expressions f_i (but not the field definitions e_i). None of the other x_i, y_i are in scope in the e_i, f_i, but you can get to them indirectly in the f_i's via m. The fields bindings are "by-value", in that the e_i are evaluated when the module expression is encountered, not when the fields are selected. The methods bindings are constrained to be functions and so are automatically values.

   The expression $e.x$ (or $e.y$) selects the field value named x (or method named y) that is exported by the module value that e evaluates to.

   The expression extend e_1 by e_2 produces a new module values from two existing ones that are obtained by evaluating e_1 and e_2. The new module value defines an identifier if it is defined in either e_1 or e_2, and the definitions in e_2 take precedence.

   Using these features, we can write code that is at least superficially object-oriented (assuming the usual desugaring for let):

```
let make_point = rec λx0.λy0.
    (module p (fields x = x0,
                      y = y0,
               methods lengthsq = λu. p.x*p.x + p.y*p.y,
                       minus    = λp2. make_point (p.x - p2.x, p.y - p2.y))) in
  let p1 = make_point 1 2 in
    let p2 = make_point 2 4 in
      p1.minus(p2).lengthsq unit
```

The result of this program is 5.

In this problem you will extend the semantics of uML to describe uML$_m$. Unlike in the module semantics given in class, modules should be treated as *environment extenders*: that is, members of the domain *Env* → *Env* that extend an existing environment with a set of possibly new bindings.

(a) Make all changes to the domain equations of uML necessary to support these language features.

(b) The module expression permits the module name to occur only in method definitions. Why is this limitation important for the ability to define a semantics for this language?

(c) Define the semantic function $\mathcal{C}$ for the new uML$_m$ expression forms.

(d) Consider the following code:

```
let o1 = module this (
           fields x = 0,
                  y = 0
           methods getx = λu. this.x
         ) in
let setx = λobj. λnewx. extend obj by module this' (
                        fields x = newx
                        methods
                      )
in
  setx(o1, 5).getx(unit)
```

What is the result of this program under your semantics? Does this agree with your intuition about how objects behave? Explain briefly.

3. Standard Semantics (25 pts)

We saw earlier that we could use CPS translation to compactly encode non-local control features. In this problem we'll use this approach to model a **break** statement like that in Java. Consider the IMP language, which already has **while**, and for which we've seen a direct semantics on the prelim:

$$
\begin{array}{lll}
x & \in & \textbf{Var} \\
a & \in & \textbf{Aexp} ::= n \;|\; x \;|\; a_1 + a_2 \;|\; a_1 - a_2 \;|\; a_1 \times a_2 \\
b & \in & \textbf{Bexp} ::= \textbf{true} \;|\; \textbf{false} \;|\; b_1 \wedge b_2 \;|\; b_1 \vee b_2 \;|\; a_1 = a_2 \;|\; a_1 \leq a_2 \\
c & \in & \textbf{Com} ::= \textbf{skip} \;|\; x := a \;|\; c_1;\; c_2 \;|\; \textbf{if}\, b\, \textbf{then}\, c_1\, \textbf{else}\, c_2 \;|\; \textbf{while}\, b\, \textbf{do}\, c
\end{array}
$$

The semantics we used on the prelim was a direct semantics; here we'll write a continuation semantics which will look very much like the CPS translations we'lve already been doing, except that the functions we will write are mathematical functions on well-defined domains. The first step is to define the domains that will be the interpretations of various terms. It makes sense to define a separate kind of continuation for each of the three kinds of terms (**Aexp**, **Bexp**, and **Com**. Recall that a continuation semantics involves continuations, which are functions that don't return a value. We introduce a domain *Answer* to represent what continuations return. Since continuations don't return, *Answer* could be equally well be almost anything. By leaving it unspecified, we ensure that we don't use it!

$$\begin{aligned} ACont &= \mathbb{Z} \to Answer \\ BCont &= \mathbb{T} \to Answer \\ CCont &= \Sigma \to Answer \end{aligned}$$

The three semantic functions then send their results to the appropriate kind of continuation:

$$\begin{aligned} \mathcal{A}[\![a]\!] &\in ACont \to \Sigma \to Answer \\ \mathcal{B}[\![b]\!] &\in BCont \to \Sigma \to Answer \\ \mathcal{C}[\![c]\!] &\in CCont \to \Sigma \to Answer \end{aligned}$$

Now we can define the meaning of the various terms. For example, the meaning of a variable expression x and a command **skip** is as follows:
$\mathcal{A}[\![x]\!] = \lambda k \in ACont\,.\,\lambda\sigma \in \Sigma\,.\,k(\sigma x)$
$\mathcal{C}[\![\textbf{skip}]\!] = \lambda k \in CCont\,.\,\lambda\sigma \in \Sigma\,.\,k\sigma$

We can even use η-reduction to simplify a bit:
$\mathcal{C}[\![\textbf{skip}]\!] = \lambda k \in CCont\,.\,k$

(a) Finish writing the rest of the continuation semantics for IMP. The **while** command will need to use *fix* just like in the direct semantics. Annotate all λ's with explicit domains, as above. You do not have to use exactly the domains given above, but if you change them you must justify it.

(b) Now suppose that we add a **break** statement to IMP. Informally, **break** causes the closest enclosing **while** statement to immediately terminate. Show how to modify your domain definitions and your semantics to support this new statement.

4. Soundness (25 pts)

We saw that the simply-typed lambda calculus has a sound type system because it preserves types and guarantees progress on well-typed terms. Thus, well-typed terms do not get stuck (evaluation is *safe*). Suppose that we add pair terms and product types as described in Lecture 24. Extend the preservation and progress proofs from $\lambda^{\to}$ to demonstrate soundness for this extended language $\lambda^{\to,*}$.