Cornell University
Computer Science 611

Problem Set 2 DUE: 1 Oct 2004

Instructions

Turn in the written part of the assignment by 5PM on the due date in Upson 4119. The programming part should be submitted using CMS (`http://cms.csuglab.cornell.edu`) by the same time. Most of the assignment is to be done individually, except for the last problem as noted below. This is a longer and more difficult problem set than PS1, so plan your time accordingly!

1. Well-founded relations (10 pts.)

   For each of the following relations, identify whether the relation is well-founded and explain why or why not.

   (a) The relation $<$ on the integers $\mathbb{Z}$.

   (b) The successor relation $\prec$ on the natural numbers ω: the relation $n \prec n+1$

   (c) A relation $\prec$ on pairs of natural numbers (n, m), where $(m, n) \prec (m', n')$ if and only if $m' = m+1$ and $n = n' + 1$.

   (d) A relation on finite trees, where given two trees t and t', $t \prec t'$ iff t is exactly the same as t' except that it is missing exactly one leaf.

   (e) A relation $\prec$ on partial functions in $\omega \rightharpoonup \omega$, where

   $$f_1 \prec f_2 \stackrel{\Delta}{\Leftrightarrow} f_1 \neq f_2 \text{ and } \mathrm{dom}(f_1) \subseteq \mathrm{dom}(f_2) \text{ and } \forall x \in \omega. f_1(x) \leq f_2(x)$$

   .

2. Proofs by induction (25 pts.)

   Consider $\mathrm{IMP}_{\mathrm{FOR}}$, a version of IMP that has **for** loops instead of **while** loops. We redefine commands c as follows:

   $$c ::= \textbf{skip} \mid x := a \mid \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1 \mid \textbf{for } x = a_0 \textbf{ to } a_1 \textbf{ do } c$$

   Let us suppose that the big-step semantics are unchanged except that we substitute the following **for** rules for the **while** rules:

   $$\frac{\langle a_0, \sigma\rangle \Downarrow n_0 \quad \langle a_1, \sigma\rangle \Downarrow n_1}{\langle \textbf{for } x = a_0 \textbf{ to } a_1 \textbf{ do } c, \sigma\rangle \Downarrow \sigma} \text{ where } n_0 > n_1$$

   $$\frac{\langle a_0, \sigma\rangle \Downarrow n_0 \quad \langle a_1, \sigma\rangle \Downarrow n_1 \quad \langle \textbf{for } x = n_0 \textbf{ to } n_1 \textbf{ do } c, \sigma\rangle \Downarrow \sigma'}{\langle \textbf{for } x = a_0 \textbf{ to } a_1 \textbf{ do } c, \sigma\rangle \Downarrow \sigma'} \text{ where } n_0 \leq n_1$$

   $$\frac{\langle c;\ \textbf{for } x = n_0' \textbf{ to } n_1 \textbf{ do } c, \sigma[x \mapsto n_0]\rangle \Downarrow \sigma'}{\langle \textbf{for } x = n_0 \textbf{ to } n_1 \textbf{ do } c, \sigma\rangle \Downarrow \sigma'} \text{ where } n_0 \leq n_1 \wedge n_0' = n_0 + 1$$

   Informally, the bounds of the loop are computed once, at the beginning of the loop, and although the loop index variable can be assigned within the loop, these assignments do not affect the value of the variable at the beginning of the next loop iteration.

   (a) Define a series of $\mathrm{IMP}_{\mathrm{FOR}}$ programs $P_1, P_2, P_3, \ldots$ such that the length of program P_n is $O(n)$ but the running times of the programs grow faster than k^n for any integer k.

   (b) Despite the fact that we can write many useful programs in this language—it can compute the *primitive recursive functions*—the language is not universal. Show that it is not universal by demonstrating that all programs terminate. You may assume that all arithmetic and boolean expressions terminate, and prove that commands terminate under this assumption. (*Hint:* Use well-founded induction, but make sure you show your well-founded relation is indeed well-founded!)

3. Operational semantics (25 pts.)

The PostScript programming language implemented by many laser printers is remarkably powerful but has some unconventional features. In this problem, you will develop a semantics for a simplified version of the language, called SubScript.

$$
\begin{array}{rcl}
x & \in & \mathbf{Var} \\
n & \in & \mathbb{Z} \\
p & ::= & c_1 \ \ldots \ c_n \\
c & ::= & x \mid /x \mid n \mid \mathsf{def} \mid \mathsf{ifp} \mid + \mid \{\ p\ \} \mid \mathsf{get}
\end{array}
$$

A program p consists of a (possibly empty) sequence of commands c. Informally, These commands each perform an operation on an (implicit) stack of values. For example, an integer command n pushes n onto the stack. A variable x that is bound to an integer pushes that integer on the stack; however, a variable bound to a block *executes* that block, replacing the variable in the command sequence with all the commands in the block. The command $/x$ pushes the *name* of the variable x on the stack. The def command expects to see on the stack a value and the name of a variable; it binds the variable to that value. The ifp command expects an integer and two blocks; if the integer is positive, it executes the first block and otherwise the second block. The + operator adds two integers. A block expression $\{p\}$ pushes that block onto the stack. And get expects the name of a variable and pushes its value onto the stack even when it is bound to a block.

Blocks are essentially functions, where invocation is *postfix* rather than *prefix* as in the lambda calculus. When a block completes execution, any variable bindings that were performed during its execution are erased and all variables are reset to the values they had previously. For example, here is a program that computes the fifth Fibonacci number. Note the recursive calls to fibo from inside the block that defines it.

```
{
    -1 +
} /pred def
{
    /n def
        n pred
        {
            n pred pred fibo
            n pred fibo
          +
        }
        {
            1
        }
    ifp
} /fibo def
5 fibo
```

(a) Do variables in this language have dynamic scope or static scope? Give an example program that shows why your answer is correct.

(b) Write a small-step operational semantics for this language. If you have to resolve any ambiguities in the textual description above, identify where this happens in your rules and why.

4. Mutual recursion and closure conversion (40 pts.)

In this problem you will demonstrate that neither mutual recursion nor nested lambda expressions are essential to the expressiveness of the lambda calculus as long as there are primitive tuple values. This is a useful result because tuples are rather easy to implement on most computers — easier than general lambda abstractions.

The file lifting.sml defines datatypes for two languages: a source language and a target language. Your job is to translate one into the other.

The source language has mutually recursive function definitions and first-class lambda expressions:

$$\begin{array}{rcl} x & \in & \textbf{Var} \\ e & ::= & x \mid \lambda x.\, e \mid e_1\ e_2 \mid \mathsf{let} f_1 = \lambda x_1.\, e_1\ \mathsf{and}\ \ldots\ \mathsf{and}\ f_n = \lambda x_n.\, e_n\ \mathsf{in}\ e_0 \\ & \mid & n \mid e_1 + e_2 \mid \mathsf{ifp}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \end{array}$$

This source language is an extension of the eager, call-by-value lambda calculus. The extension consists of the let expression, which allows several mutually recursive functions $f_1, \ldots, f_n$ to be defined; these functions may refer to one another in their respective bodies $e_1, \ldots, e_n$. To give the language something to compute on, integers and some simple operations are provided similar to those in SubScript.

The target language is more primitive, allowing functions to be declared only at the top level of a program p. However, there is a built-in tuple expression $(e_1, \ldots, e_n)$ and a selector expression $(\#ne)$ as in ML:

$$\begin{array}{rcl} p & ::= & \mathsf{let}\ f_1 = \lambda x_1.\, e_1\ \mathsf{in}\ p \mid e \\ e & ::= & x \mid e_1\ e_2 \mid (e_1, \ldots, e_n) \mid \#n\ e \\ & \mid & n \mid e_1 + e_2 \mid \mathsf{ifp}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \end{array}$$

Expressions may not introduce new lambda abstractions. In effect all lambda abstractions have been "lifted" to the top level of the program. Furthermore, the bodies of these named lambda abstractions must be closed expressions. This is an essential transformation for compiling to a low-level language. In addition to *closure conversion*, it is also known as *lambda lifting* or *environment conversion*.

Your job is to implement the function translate that converts a term in the source language into a term in the target. When the result of a program is an integer, the translated program should produce the same integer as the source program. You will probably find it easiest to implement this translation as two simpler translation steps: one that eliminates recursion, and one that lifts lambdas.

To aid you in debugging your translation, an interpreter has been included for the target language. This interpreter treats identifiers bound to functions as values, which is possible because functions may only be top-level. A useful conceptual model is that function identifiers are really code addresses.

You may do this part of the assignment (and only this part of the assignment) with a partner. However, your partner may *not* be the same person with whom you worked on the programming part of Problem Set 1 (if any). Both partners are expected to understand the complete solution. Make sure to add a comment to lifting.sml indicating who has worked on the problem, if any. This problem is a bit tricky so start early!