Cornell University
Computer Science 6110

Homework 2 DUE: Wednesday, February 26, 11:59PM

Updates

- 2/18: Partner policy clarified.

What to turn in

Turn in the assignment by midnight on CMSX on the due date, including both code and written problems.

Working with a partner

You may have a partner on this assignment. Both partners are expected to work on all parts of the assignment, and to understand the solution.

1. **Implementing lambda calculus** [40 pts]

   The file lambda.ml contains a partial implementation of some useful lambda calculus mechanisms. In particular, it contains a correct implementation of call-by-value implementation in the function cbv, and you can use it to try out evaluation. The function print_exp can be used to print a human-readable representation of an expression.

   (a) This file also includes most of the implementation of a function nf that reduces a term to $\beta\eta$-normal form, but it doesn't quite work. One of the problems is that the substitution function subst is not correct. Fix the implementation of subst, and fix any other bugs to make nf work correctly.

   (b) There is also a very incomplete implementation of a function translate that translates from an extended language to simple lambda calculus. The extended language includes let expressions (like in ML), recursive functions, and pairs. Complete translate so that it faithfully translates extended terms into lambda calculus terms.

   Complete the implementation of lambda.ml and submit the result through CMS.

2. **Well-founded relations** [20 pts]

   Which of the following relations are well-founded? Justify briefly.

   (a) Dictionary ordering on strings of alphabetic characters (a–z).

   (b) An ordering $\prec$ on pairs of natural numbers defined inductively by these rules:

   $$\frac{n_1 < n_1'}{(n_1, n_2) \prec (n_1', n_2')} \qquad \frac{n_2 < n_2'}{(n_1, n_2) \prec (n_1, n_2')}$$

   (c) An ordering $\prec$ on finite sequences of natural numbers, where a sequence s of length n is preceded by the subsequences of s and also by any sequence whose first n elements are all smaller than the corresponding elements of s.

   (d) A relation $\prec$ on partial functions in $\mathbb{N} \rightharpoonup \mathbb{N}$, where

   $$f_1 \prec f_2 \overset{\triangle}{\iff} f_1 \neq f_2 \land \mathrm{dom}(f_1) \subseteq \mathrm{dom}(f_2) \land \forall x \in \mathrm{dom}(f_1).\, f_1(x) \leq f_2(x).$$

3. **Names and scope** [10 pts]

   Consider the following program:

```
let $x$ = 5 in
    let $f$ = $\lambda y.\, x + y$ in
        let $x$ = 4 in
            let $g$ = ($\lambda z.$ let $x$ = 3 in $f(x)$) in
                $g(x) + f(x)$
```

(a) What does this program output using call-by-value semantics with static scope? Explain briefly.

(b) What does this program output using call-by-value semantics with dynamic scope? Explain briefly.

(c) What does this program output using call-by-name semantics with static scope? Explain briefly.

4. **Proving termination** [30 pts]

Consider IMP$_{\text{repeat}}$, a version of IMP that has replaces **while** loops with loops that perform a computed number of iterations. We redefine commands c as follows:

$$c ::= \textbf{skip} \mid x := a \mid \textbf{if}\ b\ \textbf{then}\ c_0\ \textbf{else}\ c_1 \mid \textbf{repeat}\ a\ \textbf{do}\ c \mid c_0;\ c_1$$

Informally, the **repeat** loop works as follows. When entering the loop **repeat**a**do**c, the expression a is evaluated to an integer n. If $n \leq 0$, the command just behaves like **skip**. If $n > 0$, the body c is executed that many times. Note that the number of iterations is computed once at the beginning of the loop, and no computation in the body of the loop can change the number of times the loop is executed.

(a) Write a big-step operational semantics for the **repeat** a **do** c construct.

(b) Write an IMP$_{\text{repeat}}$ program that given an input value in the variable x, computes in variable y the first Fibonacci number $F(n)$ (where $F(0) = 0, F(1) = 1, F(2) = 1$) such that $F(n) \geq x$. You may assume that you have multiplication, addition, subtraction, and division as built-in arithmetic operators—but not exponentiation.

(c) Despite the fact that we can write many useful programs in IMP$_{\text{repeat}}$— for example, it can compute the *primitive recursive functions* — the language is not universal. Show that it is not universal by demonstrating that all programs halt. You may assume that all arithmetic and boolean expressions halt, and you must prove that commands terminate under this assumption. If you use well-founded induction on some relation, be sure to argue that this relation is well-founded.