Advanced Programming Languages — Assignment 2
CS 6110 S11 — Due Wednesday, March 2 February 2016

Instructions: *You may complete this assignment with one other CS 6110 student. You must register with your partner on CMS before you begin working together on the assignment. With the exception of your CMS-registered partner you should not give or receive assistance on this assignment. If slip days are used, they must be used by both partners. In addition, please limit use of outside resources to the lecture notes and textbooks for this course. If you have any questions about what is allowed and what is not allowed, please contact the course staff.*

1. Observational Equivalence

   Prove Lemma 1 and Theorem 2 from Lecture 4 assuming CBV evaluation.

2. Set Operators and Fixed Points

   Exercise 4.13 Winskel (page 54).

3. Combinators

   A *combinator* is a closed λ-term of the form $\lambda x_1, \ldots, x_n . t$, where t is a term built from $x_1, \ldots, x_n$ and application. For example,

   $$\mathsf{S} \stackrel{\triangle}{=} \lambda xyz.\, xz(yz) \qquad \mathsf{K} \stackrel{\triangle}{=} \lambda xy.\, x \qquad \mathsf{I} \stackrel{\triangle}{=} \lambda x.\, x$$

   One of the mathematicians who helped develop λ-calculus, Haskell Curry, experimented with combinators as a way to eliminate the need for all the complications involving substitution in the λ-calculus. He axiomatized the behavior of combinators in terms of simple rewrite rules. For example,

   $$\mathsf{S}\ x\ y\ z\ \rightarrow\ x\ z\ (y\ z) \qquad \mathsf{K}\ x\ y\ \rightarrow\ x \qquad \mathsf{I}\ x\ \rightarrow\ x.$$

   He also proved a remarkable fact: *any* closed λ-term can be simulated by some combination of S and K.

   (a) Show that I is redundant. That is, build a combinator from S and K that behaves like $\lambda x.\, x$.

   (b) Build a combinator term from S, K, and I that behaves like $\lambda xy.\, y$

   (c) Build a combinator term from S, K, and I that does not have a normal form.

   (d) **Karma Problem:** Build a combinator term from S, K, and I that behaves like the Y combinator $\lambda f.\, (\lambda x.\, f\,(x\,x))\,(\lambda x.\, f\,(x\,x))$!

   (e) **Karma Problem:** Give a translation from λ-calculus into S-K combinators.

4. For Loops

   In this exercise we will extend IMP with for loops. The syntax for for loops is as follows,

   $$c \ ::= \ \cdots \ \mid \ \mathsf{for}\ a\ \mathsf{do}\ c$$

   where a is an arithmetic expression and c is a command. Informally, a for loop executes as follows. Upon entering the loop, the expression a is evaluated in the current state, yielding an integer n. If $n \leq 0$, the loop body is not executed at all. Otherwise, if $n > 0$, then the loop body c is executed exactly n times. No command in the body of the loop c, such as assigning to a variable, should change the number of times the loop is executed. In addition, executing the loop should not change the value of any variables that do not appear in the body c.

   (a) Write small-step operational semantics rules for this new command.

(b) Write large-step operational semantics rules for this new command.

(c) Give a careful, rigorous proof of the following fact, which states that the large-step evaluation relation for IMP, including for loops, is deterministic: $\forall c, \sigma, \sigma_1, \sigma_2.\ \langle c,\ \sigma\rangle \Downarrow \sigma_1$ and $\langle c,\ \sigma\rangle \Downarrow \sigma_2$ implies $\sigma_1 = \sigma_2$. You may use the following facts in your proof:

- $\forall a, \sigma, n_1, n_2.\ \langle a,\ \sigma\rangle \Downarrow_a n_1$ and $\langle a,\ \sigma\rangle \Downarrow_a n_2$ implies $n_1 = n_2$.
- $\forall b, \sigma, t_1, t_2.\ \langle b,\ \sigma\rangle \Downarrow_b t_1$ and $\langle b,\ \sigma\rangle \Downarrow_b t_2$ implies $t_1 = t_2$.

(d) **Karma Problem:** Consider a variant of IMP with for loops but without while loops. Prove that every program terminates.

5. Implementing IMP

The archive `imp.zip` contains a partial implementation of the IMP imperative language. We have provided all the boring infrastructure (lexer, parser, and read-eval-print loop) which should compile and run right out of the box. We have also supplied some sample IMP programs.

(a) With the starter code you can load `fact.txt` and `fib.txt`, but you cannot run them, because the evaluation functions in `eval.ml` are not implemented. Please implement them. Use the big-step operational rules as described in the lecture on IMP. There are a few extra features besides those described in the lecture, namely reading from the standard input, writing to the standard output, and some extra arithmetic operations. Once you have done this, you should be able to run `fact.txt` and `fib.txt`.

```
IMP interactive interpreter version 2016.0
>> help
Available commands are:
load <file>, list, run, help, quit
>> load etc/fact.txt
>> run
? 5
120
>> list
n := input;
f := 1;
while (n > 1) {
  f := f * n;
  n := n - 1;
}
print f;
>> quit
bye
```

(b) Add the `for` loop for which you wrote SOS rules. Again, use the big-step rules. You will also have to add this to the lexer and parser, but you can use the `while` statement for a model, since the concrete syntax is very similar; see the files `fact2.txt` and `fib2.txt` for examples. Modify the specification files `lexer.mll` and `parser.mly`, and use OCamllex and OCamlyacc to build the lexer and parser. See the supplied Makefile for the appropriate incantations.

6. Debriefing

(a) How many hours did you spend on this assignment?

(b) Would you rate it as easy, moderate, or difficult?

(c) Did everyone in your study group participate?

(d) How deeply do you feel you understand the material it covers (0%–100%)?

(e) If you have any other comments, we would like to hear them! Please write them here or send email to `jnfoster@cs.cornell.edu`.