Advance Programming Languages — Assignment 1

CS 6110 Spring 2010 — Due Friday, 12 February 2016

Instructions: *You may complete this assignment with another 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. 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. Free and Bound Variables

   (a) Write each of the following λ-terms in fully-parenthesized, curried form. Then, circle each free variables and draw a box around each bound variable. Finally, for each bound variable, *neatly* draw an arrow pointing to its binder.

   (i) $\lambda abc.\, a\, a\, (\lambda b.\, b)\, b\, c\, c$

   (ii) $(\lambda x.\, x\, x)\, \lambda x.\, x\, x$

   (iii) $\lambda x.\, (\lambda y.\, z\, y)\, \lambda z.\, z\, y\, x$

   (b) Fully reduce $(\lambda x.\, \lambda y.\, x\, y)\, ((\lambda x.\, \lambda z.\, x\, z)\, (\lambda y.\, y))$ by repeatedly applying the β-reduction rule. Show each step of reduction, and underline the redex at each step.

2. Safe Substitution

   The rules for safe substitution contain a number of side-conditions of the form $y \notin FV(e)$ whose purpose may not be immediately clear. For each such side-condition, find a counterexample demonstrating that the condition is necessary to avoid variable capture. Each of your counterexamples should either converge to different normal forms under safe substitution and substitution without the condition, or should converge under safe substitution and diverge under substitution without the condition (or vice versa). Please use the definition given in the Lecture 2 notes.

3. Small-Step Operational Semantics

   Full β reduction is a reduction strategy where the β-reduction rule may be applied to any redex appearing in the term (including redexes under a λ-binder). Formalize this reduction strategy by giving a complete small-step operational semantics for it.

4. λ-Calculus Encodings

   There are many other ways to encode the natural numbers into the λ-calculus besides the Church encoding we saw in lecture. Here is one:

$$\begin{aligned}\widehat{0} &\triangleq \lambda f x.\, x\\ \widehat{1} &\triangleq \lambda f x.\, f\ \widehat{0}\ x\\ \widehat{2} &\triangleq \lambda f x.\, f\ \widehat{1}\ x\end{aligned}$$

   (a) Write a function succ that takes a number and computes its successor. You may assume the CBV reduction strategy.

   (b) Write a function pred that takes a number and computes its predecessor. Your function should map $\widehat{0}$ to $\widehat{0}$.

   (c) Write a function iszero that takes a number and tests whether it is equal to zero. Your function should return a Church boolean.

(d) Write a function sum that takes a number and computes the sum of all numbers less than it. Your function should map $\widehat{0}$ to $\widehat{0}$.

(e) **Karma problem:** Recall the Church encoding of the natural numbers $\mathbb{N}$ described in class:

$$\begin{aligned} \overline{0} &\triangleq \lambda f.\, \lambda x.\, x \\ \overline{1} &\triangleq \lambda f.\, \lambda x.\, f\ x \\ \overline{2} &\triangleq \lambda f.\, \lambda x.\, f\ (f\ x) \end{aligned}$$

Write the pred function for Church-encoded numbers.

5. λ-Calculus Programming

(a) Write a λ-term curry that converts a function that takes inputs of the form (x, y) and produces a function that does the same thing, but takes its arguments one at a time. (You may assume that we have extended λ-calculus with built-in pairs and the usual projectionoperators.)

(b) Write a λ-term uncurry that does the conversion in the opposite direction. For the uncurried function, you need not worry about inputs that are not of the form (x, y).

6. Implementing the λ-Calculus

The archive `lambda.zip` contains a partial implementation of a λ-calculus interpreter. The syntax recognized by the parser is based on OCaml. For example, $\lambda xy.\, e$ is written `fun x y -> e`.

The interpreter's main function is a read-eval-print loop that reads in a sequence of lines from the terminal (terminated by a blank line), parses and desugars the input term, prints it, and then steps through its CBV evaluation, showing one step each time the user hits enter.

```
This program computes the translation and the step by step evaluation
of an expression. Type in an expression, then type <Enter> twice.
? (fun x y -> x) (fun z -> z) (fun u v -> u)

Translation: (fun x -> (fun y -> x)) (fun z -> z) fun u -> fun v -> u
>>
(fun y -> (fun z -> z)) fun u -> fun v -> u
>>
Result: fun z -> z
```

Your task is to complete the implementation of the interpreter:

(a) The construct `let x = e1 in e2` is syntactic sugar for `(fun x -> e2) e1`. Similarly, the construct `fun x1 x2 ... xn -> e` is sugar for `fun x1 -> fun x2 -> ... -> fun xn -> e`. Expressions of type `expr_s` contain constructors to represent these extensions, while expressions of type `expr` do not. Complete the implementation of the desugaring function `translate`.

(b) Implement the function `subst` that substitutes an expression `v` for a variable `x` in an expression `e`. Be extremely careful to avoid variable capture!

(c) Complete the implementation of `cbv_step`.

You only need to modify the `lambda.ml` file where indicated and should not need to introduce any new functions. Some helper functions can be found in `ast.ml` and `util.ml`.

7. 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`.