CS 6110 Spring 2017 — Assignment 1

Advanced Programming Languages — Due: Wednesday, 15 February 2017

Instructions: You may complete this assignment with another CS 6110 student. You must form a group with your partner on CMS before you begin working together on the assignment. With the exception of the course staff and your CMS partner, you should not give or receive assistance on this assignment. In addition, please limit use of resources to the lecture notes for this course. If you have any questions about what is allowed and what is not allowed, please contact the course staff.

1. β-reduction warm-up.

Consider the closed λ-term $(\lambda x.\, \lambda y.\, x\, y)\, ((\lambda x.\, \lambda z.\, x\, z)\, (\lambda y.\, y))$.

(a) Fully reduce the term using *call-by-value* (CBV) evaluation. Show each step in the β-reduction.

(b) Fully reduce the term in *call-by-name* (CBN) order, again showing each step.

2. Capture-avoiding substitution.

These rules define safe substitution in the λ-calculus:

$$
\begin{array}{rcll}
x\{e/x\} & \triangleq & e & \\
y\{e/x\} & \triangleq & y & \text{where } y \neq x \\
(e_1\, e_2)\{e/x\} & \triangleq & (e_1\{e/x\})\, (e_2\{e/x\}) & \\
(\lambda x.\, e_0)\{e/x\} & \triangleq & \lambda x.\, e_0 & \\
(\lambda y.\, e_0)\{e/x\} & \triangleq & \lambda y.\, (e_0\{e/x\}) & \text{where } y \neq x \text{ and } y \notin FV(e) \\
(\lambda y.\, e_0)\{e/x\} & \triangleq & \lambda z.\, (e_0\{z/y\}\{e/x\}) & \text{where } y \neq x,\ z \neq x,\ z \notin FV(e_0), \text{ and } z \notin FV(e)
\end{array}
$$

In this exercise, you will justify the need for those *side conditions* to the right of the last two rules.

For each of the conditions regarding free variables, write a *counter-example* that shows why the condition is necessary. Specifically, write a λ-term that can β-reduce to the wrong result if the condition is removed from the definition of substitution.

For each part, write your counter-example term, a value it reduces to under *correct* substitution, and a "wrong" value it can reduce to under the substitution definition with the condition dropped.

(a) $y \notin FV(e)$ in the second-to-last rule.

(b) $z \notin FV(e_0)$ in the last rule.

(c) $z \notin FV(e)$ in the last rule.

3. Operational semantics.

In class, we saw the inference rules that define the small-step operational semantics for *call-by-value* (CBV) and *call-by-name* (CBN) evaluation of λ-calculus terms. For example, here is the semantics for CBV evaluation:

$$
\frac{}{(\lambda x.\, e)\, v \xrightarrow{1} e\{v/x\}} \qquad \frac{e_1 \xrightarrow{1} e_1'}{e_1\, e_2 \xrightarrow{1} e_1'\, e_2} \qquad \frac{e \xrightarrow{1} e'}{v\, e \xrightarrow{1} v\, e'}
$$

We also informally discussed *full β-reduction*, where evaluation allows any redex in the term to be reduced—even redexes that appear "under a lambda." Full β-reduction, unlike CBV and CBN, is nondeterministic: multiple steps are possible from any given term.

Write inference rules for the small-step operational semantics of the λ-calculus under full β-reduction.

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:

$$\widehat{0} \triangleq \lambda f x.\, x$$
$$\widehat{1} \triangleq \lambda f x.\, f\ \widehat{0}\ x$$
$$\widehat{2} \triangleq \lambda f x.\, f\ \widehat{1}\ (f\ \widehat{0}\ x)$$
$$\vdots$$

These exercises ask you to write encodings for functions on these numbers. You may assume call-by-value evaluation.

(a) Write a function succ that takes the representation of a number using this technique, i.e., $\widehat{n}$ for any natural number n, and computes its successor, i.e., $\widehat{n+1}$.

(b) Write a function pred that takes an encoded number $\widehat{n}$ and computes its predecessor, $\widehat{n-1}$. Your function should map $\widehat{0}$ to $\widehat{0}$.

(As an aside, the predecessor function is very hard—but possible!—to write using standard Church numerals. This funky encoding makes it easier.)

(c) Write a function iszero that takes a number and tests whether it is equal to zero. Your function should return a Church Boolean: either true, a.k.a. $\lambda xy.\, x$, or false, a.k.a. $\lambda xy.\, y$. (You may use the names true and false in your solution.)

5. λ-calculus programming.

Consider a version of the λ-calculus extended with pairs. The syntax is extended with three new forms:

$$e \quad ::= \quad \cdots \quad | \quad (e_1, e_2) \quad | \quad \mathsf{first}\ e \quad | \quad \mathsf{second}\ e$$

These three expression forms create a pair, get the first element of a pair, and get the second element of a pair. In this version of the λ-calculus, you can write functions that take two arguments at once in the form of a pair. For example, $\lambda p.\, (\mathsf{first}\ p)\ (\mathsf{second}\ p)$ takes a pair (f, x) as an argument and applies f to x.

(a) Write a λ-term `curry` that converts a function that takes a pair as an argument and produces a function that does the same thing but takes its arguments one at a time.

(b) Write a λ-term `uncurry` that does the conversion in the opposite direction. (The uncurried function does not need to do anything in particular when its argument is not a pair.)

6. Implementing the λ-calculus.

The archive `lambda.zip` contains a partial implementation of a λ-calculus interpreter in OCaml. The archive also contains a README with instructions for how to install OCaml and how to build the interpreter.

The interpreter's OCamlesque syntax uses `fun` and `->` as the syntax for writing functions. For example, $\lambda xy.\, e$ is written `fun x y -> e`.

The interpreter's main function is a read-evaluate-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.

```
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)

(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 curried function `fun x1 x2 ... xn -> e` is sugar for the nested functions `fun x1 -> fun x2 -> ... -> fun xn -> e`. The source code defines two data types: `expr_s` represents expressions that may contain the expressions above, while `expr` represents plain λ-calculus expressions.

Complete the implementation of the desugaring function `translate` in `lambda.ml`.

(b) Implement the function `subst` that substitutes an expression `v` for a variable `x` in an expression `e`. Substitution should avoid variable capture.

(c) Complete the implementation of `cbv_step`, which β-reduces a term by a single step using the call-by-value reduction order.

All of these functions have short implementations. You can use the utility functions from `ast.ml` and `util.ml`. For example, there is an `fv` function that produces the set of free variables for an expression. That function returns a `HashSet` of variable names; use `HashSet.mem` to check whether it contains a specific name. There is also a `fresh` function that invents a new, unused variable name.

You only need to modify `lambda.ml`; please submit this file on CMS.

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 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 `asampson@cs.cornell.edu`.