Computer Language Processing

Exercise Sheet 05

October 26, 2022

Welcome to the fifth exercise session of CS320!

Exercise 1

Consider an expression language with a halving operator on even numbers. We are designing an operational semantics and a type system that ensures that we never halve an odd number.

- expr ::= half(expr) | expr + expr | INTEGER

The values of our language are all integers. We denote values by n and k.

We will design the operational semantics of our language. Semantics should define rules that apply to as many expressions as possible subject to the following constraints:

- Our operational semantics should not permit halving unless the value of an integer constant is even
- It should only perform evaluation of operands from left to right

Find a minimal subset of the operational semantics rules below that describe this behavior

$$\frac{e \rightsquigarrow e'}{\text{half}(e) \rightsquigarrow e'} \tag{A}$$

$$\frac{n \text{ is a value} \qquad n \ = \ 2k}{\text{half}(n) \rightsquigarrow k} \tag{B}$$

$$\frac{n \text{ is a value}}{\text{half}(n) \rightsquigarrow \lfloor \frac{n}{2} \rfloor} \tag{C}$$

$$\frac{\text{half}(e) \rightsquigarrow \text{half}(e')}{\text{half}(e) \rightsquigarrow e'} \tag{D}$$

$$\frac{e \rightsquigarrow e'}{\text{half}(e) \rightsquigarrow \text{half}(e')} \tag{E}$$

$$\frac{e' \rightsquigarrow \mathrm{half}(e)}{\mathrm{half}(e) \rightsquigarrow e'} \tag{F}$$

$$\frac{n_1 \text{ is a value} \qquad n_2 \text{ is a value} \qquad k = n_1 + n_2 \qquad n_1 \text{ is odd}}{n_1 + n_2 \rightsquigarrow k} \tag{G}$$

$$\frac{e \rightsquigarrow e' \qquad n \text{ is a value}}{n + e \rightsquigarrow n + e'} \tag{H}$$

$$\frac{e_2 \rightsquigarrow e_2'}{e_1 + e_2 \rightsquigarrow e_1 + e_2'} \tag{I}$$

$$\frac{n_1 \text{ is a value} \qquad n_2 \text{ is a value} \qquad k = n_1 + n_2 \qquad n_1 \text{ is even} \quad n_2 \text{ is even}}{n_1 + n_2 \rightsquigarrow k} \tag{J}$$

$$\frac{n_1 \text{ is a value} \qquad n_2 \text{ is a value} \qquad k = n_1 + n_2}{n_1 + n_2 \rightsquigarrow k} \tag{K}$$

$$\frac{e_1 \rightsquigarrow e_1'}{e_1 + e_2 \rightsquigarrow e_1' + e_2} \tag{L}$$

Exercise 2

Consider a simple programming language with integer arithmetic, boolean expressions and user-defined functions.

$$
\begin{aligned}
t \; := \; & true \mid false \mid c \\
& \mid t == t \mid t + t \\
& \mid t \;\&\&\; t \mid if\ (t)\ t\ else\ t \\
& \mid f(t, ..., t) \mid x
\end{aligned}
$$

Where c represents integer literals, $==$ represents equality (between integers, as well as between booleans), $+$ represents the usual integer addition and $\&\&$ represents conjunction. The meta-variable f refers to names of user-defined function and x refers to names of variables. You may assume that you have a fixed environment e which contains information about user-defined functions (i.e. the function arguments and the function body).

1) Inductively define the substitution operation for your terms, which replaces every free occurrence of a variable in an expression by an expression without free variables.

The rule for substitution in an addition is provided as an example. Here, $t[x \; := e]$ denotes the substitution of every free occurrence of x by e in t.

$$
\frac{t_1[x \; := e] \rightarrow t_1' \qquad t_2[x \; := e] \rightarrow t_2'}{(t_1 + t_2)[x \; := e] \rightarrow (t_1' + t_2')}
$$

2) Write the operational semantics rules for the language, assuming call-by-name semantics for function calls. In call-by-name semantics, the arguments of a function are not evaluated before the call. In your operational semantics, parameters in the function body are to merely be substituted by the corresponding unevaluated argument expression.

Exercise 3

Consider the following grammar and evaluation rules for untyped lambda calculus with call-by-value semantics:

- Values: v ::= $\lambda x.\ t_1$
- Terms: t ::= x | $\lambda x.\ t_1$ | t t (left-associative)

$$\frac{t_1 \rightsquigarrow t_1'}{t_1\ t_2 \rightsquigarrow t_1'\ t_2} \qquad (\textsc{App1})$$

$$\frac{t_2 \rightsquigarrow t_2'}{v\ t_2 \rightsquigarrow v\ t_2'} \qquad (\textsc{App2})$$

$$(\lambda x.\ t_1)\ v \rightsquigarrow t_1[x \mapsto v] \qquad (\textsc{AppAbs})$$

We use Church encoding to represent numbers. In particular, we define the following terms:

$$\begin{aligned}
c_0 &= \lambda s.\ \lambda z.\ z \\
c_1 &= \lambda s.\ \lambda z.\ s\ z \\
c_2 &= \lambda s.\ \lambda z.\ s\ (s\ z) \\
c_3 &= \lambda s.\ \lambda z.\ s\ (s\ (s\ z)) \\
plus &= \lambda m.\ \lambda n.\ \lambda s.\ \lambda z.\ m\ s\ (\ n\ s\ z) \\
scc &= \lambda n.\ \lambda s.\ \lambda z.\ s\ (n\ s\ z)
\end{aligned}$$

1) Evaluate the expression $(succ\ (succ\ c_1))$ step-by-step following the semantics above
2) Make minimal changes to the evaluation rules in order to match the call-by-name semantics. With call-by-name semantics, a function can be applied to its argument even if the argument is not completely reduced (ie. it is not a value). Repeat the evaluation of $(succ\ (succ\ c_1))$ with the new semantics.
3 As you might notice, we are still not able to obtain the reduced form $c_3 = \lambda s.\ \lambda z.\ s\ (s\ (s\ z))$ after evaluation with the call-by-name semantics. Add a new rule (to either call-by-value or call-by-name) that would allow us to obtain such a form. Evaluate $(succ\ (succ\ c_1))$ to verify your answer.