Semantics Engineering: An Example

August 3, 2015

This section works through the development of a semantics for a simple functional language to illustrate the process of semantics engineering along with reduction semantics, the approach to modeling that Redex is designed to most easily take advantage of.

Figure 1 shows the grammar for the language we'll be modeling in this section. It is a language of numbers and functions, with two binary operations on numbers, addition (+) and subtraction (-), along with a conditional (if0) that dispatches on whether or not its first argument evaluates to 0 or not. Expressions beginning with λ construct functions of a single argument, which are applied via parenthesized juxtaposition as in Racket or other languages in the LISP family. Finally, rec expressions support the construction of recursive bindings.

A semantics for a programming language is a function from programs to answers. The kind of function used and what is meant by answers varies, depending on what the semantics is to be used for. Here we will develop an operational semantics in the form of a syntactic machine that transforms programs until they become answers.

To develop a semantics for this language, we start by identifying answers, a subset of expressions that are *values*, the results or final states of *computations*. For this language the right choices are numbers and functions, both of which cannot be further evaluated without being used in another expression. We denote values with the addition of another nonterminal, v:

$$v ::= (\lambda\ [x\ \tau]\ e) \mid n$$

$$\begin{array}{l} e ::= (e\ e) \\ \quad \mid (\lambda\ [x\ \tau]\ e) \\ \quad \mid (\mathsf{rec}\ [x\ \tau]\ e) \\ \quad \mid (\mathsf{if0}\ e\ e\ e) \\ \quad \mid (o\ e\ e) \\ \quad \mid x \mid n \\ n ::= \textit{number} \\ o ::= + \mid - \\ x ::= \textit{variable-not-otherwise-mentioned} \end{array}$$

Figure 1: Grammar for expressions.

We expect that all valid programs (more will be said below about what that means) either are a value, or will eventually evaluate to a value.

To this end, we develop a set of rules known as a notion of reduction. The rules are directed relations, pairing any expression in the language that is not a value with another expression that is in some sense "closer" to being a value. ("Closer" in this sense is usually a fairly intuitive, but in the end it is necessary to prove that a semantics based on these rules does the right thing by eventually transforming all valid programs into values.) For example, the notion of reduction for our binary operations looks like:

$$(o\ n_1\ n_2) \longrightarrow \ulcorner n_1\ o\ n_2 \urcorner\ [\delta]$$

Meaning that when a binary operation is applied to two numbers in an expression, we can relate that expression to the number that is the result the corresponding operation on numbers. This allows us to "reduce" such a binary operation to a value. A simple example is:

$$(+\ 1\ 2) \longrightarrow 3$$

The rule for function application is more interesting. It says that when a function is applied to a value, the resulting expression is constructed by substituting the value v for all instances of the variable x bound by the function in the function's body e:

$$((\lambda\ [x\ \tau]\ e)\ v) \longrightarrow e\{x \leftarrow v\}\ [\beta]$$

Where the notation $e\{x \leftarrow v\}$ means to perform *capture-avoiding* substitution of e for x in v. For example, the application of a function that adds one to its argument to two reduces as follows:

$$((\lambda\ (x\ \mathsf{num})\ (+\ 1\ x))\ 2) \longrightarrow (+\ 1\ 2)$$

The complete set of reductions adds rules for $\mathsf{if0}$ and rec and is shown in figure 2.

$$((\lambda\ [x\ \tau]\ e)\ v) \longrightarrow e\{x \leftarrow v\} \qquad [\beta]$$

$$(\mathsf{rec}\ [x\ \tau]\ e) \longrightarrow e\{x \leftarrow (\mathsf{rec}\ [x\ \tau]\ e)\} \qquad [\mu]$$

$$(\mathsf{if0}\ 0\ e_1\ e_2) \longrightarrow e_1 \qquad [\text{if-0}]$$

$$(\mathsf{if0}\ n\ e_1\ e_2) \longrightarrow e_2 \qquad [\text{if-n}]$$
$$\text{where } n \neq 0$$

$$(o\ n_1\ n_2) \longrightarrow \ulcorner n_1\ o\ n_2 \urcorner \qquad [\delta]$$

Figure 2: The complete set of reductions for this language.