CS603 Organization of Programming Languages

University of Alabama
Department of Computer Science
Spring 2005

From Section 2.4.1 of Ramsey and Kamin

The transition rules of the abstract machine are written in the form of *judgments*. There is one kind of judgment for top-level items and a different kind of judgment for expressions. We write the judgement $\langle e, \xi, \phi, \rho\rangle \Downarrow \langle v, \xi', \phi, \rho'\rangle$ to mean "evaluating expression e produces value v." More precisely, the judgment means "in the environment ξ, ϕ, and ρ, evaluating e produces a value v, and it also produces new environments ξ' and ρ', while leaving ϕ unchanged." We always use symbols e and e_i for expressions, v and v_i for values, and x and x_i for names.

By looking at the *form* of the judgment $\langle e, \xi, \phi, \rho\rangle \Downarrow \langle v, \xi', \phi, \rho'\rangle$, we can already learn a few things:

- Evaluating an expression always produces a value, unless of course the machine gets stuck. Even expressions like SET and WHILE, which are typically evaluated only for side effects, must produce values.
- Evaulating an expression might change the value of a global variable, (from ξ) or a formal parameter (from ρ).
- Evaulating an expression never adds or changes a function definition (becuase ϕ is unchanged).

One thing we *can't* learn from the form of the judgment is whether evaluating an expression can introduce a new variable. In fact it can't, but to learn this requires that we study the full semantics and write an inductive proof.

The heart of the interpreter in Section 2.4 is the function `eval`. Calling $\texttt{eval}(e, \xi, \phi, \rho)$ return v and has side effects on ξ and ρ such that $\langle e, \xi_{before}, \phi, \rho_{before}\rangle \Downarrow \langle v, \xi_{after}, \phi, \rho_{after}\rangle$. The rules in this section specify a straightforward, recursive implementation of `eval`.

We have a simpler kind of judgment for top-level items. We write $\langle t, \xi\phi\rangle \rightarrow \langle \xi', \phi'\rangle$ to mean "evaluating top-level item t in the environments ξ and ϕ yields new environments ξ' and ϕ'."

We can't write just any judgment and expect it to be true. For example, it seems reasonable to claim $\langle (+\ 1\ 1), \xi, \phi, \rho\rangle \Downarrow \langle 2, \xi, \phi, \rho\rangle$, but $\langle (+\ 1\ 1), \xi, \phi, \rho\rangle \Downarrow \langle 4, \xi, \phi, \rho\rangle$ is clearly nonsense. An operational semantics uses *rules of inference* to tell which judgments are valid. Each rule has the form

$$\frac{\text{premises}}{\text{conclusion}}\ \textsc{Name of Rule}$$

If we can find a way to prove each premise, we can use the rule to prove the conclusion.

For example, the rule

$$\frac{\langle e_1, \xi, \phi, \rho\rangle \Downarrow \langle v_1, \xi', \phi, \rho'\rangle \quad v_1 \neq 0 \quad \langle e_2, \xi', \phi, \rho'\rangle \Downarrow \langle v_2, \xi'', \phi, \rho''\rangle}{\langle \text{IF}(e_1, e_2, e_3), \xi, \phi, \rho)\rangle \Downarrow \langle v_2, \xi'', \phi, \rho''\rangle}\ \textsc{IfTrue}$$

which is part of the semantics of Impcore, says that whenever $\langle e_1, \xi, \phi, \rho\rangle$ evaluates to some nonzero value v_1, the expression $\text{IF}(e_1, e_2, e_3)$ evaluates to the result of evaluating e_2. Because e_2 is evaluated in the environment

produced by evaluation of e_1, if e_1 contains side effects, such as assigning to a variable, the results of those side effects will be visisble to e_2. The premises don't even mention e_3 because if $v_1 \neq 0$, e_3 is never evaluated.

The recursive implementation of `eval` actually works *bottom-up* through rules. It is given a state $\langle e, \xi, \phi, \rho \rangle$, and it finds a new state $\langle v, \xi', \phi, \rho' \rangle$ such that $\langle e, \xi, \phi, \rho \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle$. It does so by looking at the *form* of e and examining rules that have e's of that form in their conclusions. For example, to evaluate an IF expression, it first makes a recursive call to itself to find v_1, ξ', and ρ' such that $\langle e_1, \xi, \phi, \rho \rangle \Downarrow \langle v_1, \xi', \phi, \rho' \rangle$. Then, if $v_1 \neq 0$, it can make another recursive call to find v_2, ξ'', and ρ'' such that $\langle e_2, \xi', \phi, \rho' \rangle \Downarrow \langle v_2, \xi'', \phi, \rho'' \rangle$. Having satisfied all the premises of rule IfTrue, it can then return v_2 and the modified environments.

Let's work through an example. Let's evaluate the top-level `(val x 1)`. The corresponding Abstract Syntax Tree (AST) form for this is VAL(x, LITERAL(1)). Given this form, how do we go about "firing" the appropriate judgments? The easiest way is to do post-order traversal of the tree. For our given example, this means that before evaluating VAL(...), we must evaluate its child, LITERAL(1), first. To do this we evaluate LITERAL(1) in the initial environment. In the initial environment, the function environment ϕ contains a binding from the names of primitives to a term of PRIMITIVE. The global value environment, ξ, is empty. There is, of course, no parameter environment, ρ.

The rule for dealing with LITERAL is:

$$\frac{}{\langle \text{LITERAL}(v), \xi, \phi, \rho) \rangle \Downarrow \langle v, \xi, \phi, \rho \rangle} \textsc{Literal}$$

which converts the textual version of the literal ("1") into the value 1. The rule for dealing with VAL:

$$\frac{\langle e, \xi, \phi, \{\} \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle}{\langle \text{VAL}(x, e), \xi, \phi \rangle \rightarrow \langle \xi'\{x \mapsto v\}, \phi \rangle} \textsc{DefineGlobal}$$

This is something of a hack, because it evaluates the expression e as if it was executed outside the top-level, i.e. within a function. That is indicated by the $\Downarrow$. This "pseudo-function" has no arguments, as indicated by the empty local environment ("{}"). The result of evaluating the expression `(val x 1)` using the judgments Literal and DefineGlobal is to create a binding in the global value environment from x to the value 1, $\{x \mapsto 1\}$.