Formal Programming Language Semantics note 5

A language with run-time errors

The language **IMP** defined in Notes 3 and 4 serves to illustrate the basic technique of structural operational semantics. However, **IMP** is not a very interesting language in that there is really only one reasonable semantics for it. (One can of course think of minor variations on the definition we have given, but most of these end up defining exactly the same evaluation relation.) Thus, it does not serve to illustrate any of the interesting semantic choices that arise in language design, or the ways in which a formal semantics can capture these.

In this and the next few notes, we will extend and mutilate the language **IMP** in various ways. This will allow us to examine some of the semantic choices that can arise and the difference these choices make; it will also give us an opportunity to extend our repertoire of techniques for giving language definitions in an operational style.

Dynamic scoping. In **IMP**, we were concerned with programs involving a fixed set of program variables to which an initial value had already been assigned. We will now consider a variant **IMP**^{e} in which locations for variables are not allocated until a value is assigned to them — this brings us a step closer to realism, and means that we can allow an infinite set of identifiers if we want. In **IMP**^{e}, we don't have to explicitly *declare* the variables we want to use — a memory cell will automatically be allocated to a variable X the first time we attempt to assign to X. If we attempt to read the variable before we have assigned to it, a run-time error will result, specifying the identifier in question. (In these respects, **IMP**^{e} behaves like the version of BASIC I learned when I was at school. It is also like the way macros in LaTeX behave, as I am constantly being reminded while typesetting these notes!)

In general, we say a variable is *in scope* at a certain point in the execution of a program if it exists (i.e. has been defined) at that point. In **IMP**^{e}, the set of variables in scope at a given time is determined simply by which variables happen to have been assigned in the course of the execution. Note that it is possible for a variable X to be in scope at a certain point in a program even if no assignment to X occurs before this point in the program text. [Exercise: give an example to illustrate this.] We therefore say that **IMP**^{e} has *dynamic scoping* — that is, which variables are in scope at a certain point is not determined until run-time.

In a dynamically scoped language, it will in general be *undecidable* whether or not all the variables occurring in the program will always be in scope whenever they are evaluated. We therefore cannot expect a compiler to be able to check in all cases whether a program is "well-scoped" — we are therefore forced to cater for the possibility that programs will attempt to evaluate an out-of-scope variable. Something like run-time errors are therefore inevitable.

Dynamic scoping contrasts with the *static scoping* of languages like Java and ML, in which the scope of variables can be determined at compile-time from the textual structure of the program. (The most common kind of static scoping is *lexical scoping*, in which the scope of variables is determined in a simple way by the block structure of programs.) We will consider the semantics of statically scoped languages in a later note.

Definition of IMP^e. The syntax of **IMP**^{e} is identical to that of **IMP**. However, we will need to modify the semantics in order to say what happens when not all the variables have been allocated. Firstly, we will need to change the definition of state: a state for **IMP**^{e} should now be a function $\sigma : \mathbb{I} \to (\mathbb{Z} \cup \{?\})$, where $?$ is a special value signalling "undefined". We will write $\mathbb{S}_?$ for the set of states in this new sense. (Equivalently, we could take states to be *partial* functions from $\mathbb{I}$ to $\mathbb{Z}$; this is really just a question of taste and there is not much to choose between them.)

Secondly, we need to change the set of possible *results* of running a computation to take account of the possibility of run-time errors. If we attempt to read an unassigned variable X, we would like an error to be raised giving us the name of X. Let us therefore define a set of *error values* $\mathbb{E}$ by

$$\mathbb{E} \;=\; \{\mathit{undef}(X) \mid X \in \mathbb{I}\}.$$

We will use $e, e', \ldots$ as variables ranging over $\mathbb{E}$.

The result of evaluating an arithmetic expression will either be a normal integer or an error value, so let us write $\mathbb{Z}^e$ for the set $\mathbb{Z} \sqcup \mathbb{E}$. Similarly, evaluating a boolean expression can give either a truth value or an error value, so let us take $\mathbb{T}^e = \mathbb{T} \sqcup \mathbb{E}$. Finally, the execution of a command can either terminate normally yielding a final state, or can terminate prematurely if an error arises, so let us take $\mathbb{S}^e_? = \mathbb{S}_? \sqcup \mathbb{E}$. Summarizing all this, the evaluation relation for **IMP**^{e} will be some subset

$$E \;\subseteq\; (\mathsf{Aexp} \times \mathbb{S}_? \times \mathbb{Z}^e) \;\sqcup\; (\mathsf{Bexp} \times \mathbb{S}_? \times \mathbb{T}^e) \;\sqcup\; (\mathsf{Com} \times \mathbb{S}_? \times \mathbb{S}^e_?).$$

(Of course, one might also be interested in saying what the final state should be even when an error has arisen, but we shall make life easier for ourselves by not being interested in this.)

We now have to modify the semantic rules given in Note 4. The rule for evaluating a variable (2) will need to be replaced by two new rules:

$$(2.1e) \qquad \frac{}{\langle X, \sigma\rangle \Downarrow n} \; \sigma(X) = n$$

$$(2.2e) \qquad \frac{}{\langle X, \sigma\rangle \Downarrow \textit{undef}(X)} \; \sigma(X) = ?$$

All the remaining rules in Note 4 (including the definition of $\sigma[X \mapsto n]$ and the rule for assignment (14)) are still correct for describing normal (i.e. error-free) computations, and can be left unchanged.

The exception convention. Finally, we need to add rules to ensure that if an error arises in the course of running a program, it is *propagated* so that the result of executing the whole program is the corresponding error value. A moment's reflection shows that we will need to add a lot of extra rules in order to achieve this. For instance, consider rule (14) from the definition of **IMP**:

$$\frac{\langle b, \sigma\rangle \Downarrow \textit{true} \qquad \langle c_0, \sigma\rangle \Downarrow \sigma'}{\langle \texttt{if } b \texttt{ then } c_0 \texttt{ else } c_1, \sigma\rangle \Downarrow \sigma'}$$

This rule tells us how what a normal execution of a conditional looks like (when the condition is true), but we also have to take account of the possibility of an error arising, either in the evaluation of the condition b or in the execution of c_0. This means that in addition to the above rule (14) we need two further rules:

$$(14.1e) \qquad \frac{\langle b, \sigma\rangle \Downarrow e}{\langle \texttt{if } b \texttt{ then } c_0 \texttt{ else } c_1, \sigma\rangle \Downarrow e}$$

$$(14.2e) \qquad \frac{\langle b, \sigma\rangle \Downarrow \textit{true} \qquad \langle c_0, \sigma\rangle \Downarrow e}{\langle \texttt{if } b \texttt{ then } c_0 \texttt{ else } c_1, \sigma\rangle \Downarrow e}$$

In fact, we will need to add propagation rules like this for *every* existing rule in the semantics with at least one premise. However, rather than writing out endless lists of boring propagation rules, we can simply describe the general way in which these rules are obtained from the existing ones. In general, for each rule whose form (ignoring side-conditions) is

$$\frac{\langle P_1, \sigma_1\rangle \Downarrow R_1 \quad \cdots \quad \langle P_k, \sigma_k\rangle \Downarrow R_k}{\langle P, \sigma\rangle \Downarrow R}$$

and for each $1 \le i \le k$, we add an extra rule of the form

$$\frac{\langle P_1, \sigma_1\rangle \Downarrow R_1 \quad \cdots \quad \langle P_i, \sigma_i\rangle \Downarrow e}{\langle P, \sigma\rangle \Downarrow e} \; R_1, \ldots, R_{i-1} \notin \mathbb{E}$$

where e is a metavariable ranging over $\mathbb{E}$ and not occurring in the original rule. This way of specifying the necessary propagation rules is known as the *exception* *convention*; it is useful for describing how exceptions propagate in ML or Java, for instance.

Having given the semantic rules for **IMP**^{e}, the evaluation relation is defined by induction in the usual way. This completes our definition of the language **IMP**^{e}.

[Exercise: Devise an additional language construct for trapping exceptions, similar to `handle` in ML or `catch` in Java. Give a suitable semantic rule for this construct.]

Semantic choices manifested by errors. When we gave the semantics of **IMP** we made certain choices regarding the form of the rules, but we did not bother to draw attention to these choices, since typically in **IMP** they did not make any difference to the evaluation relation. Some of these choices concern *evaluation order*. For example, we could have replaced rule (3) for subtraction by the following rule, which suggests that we intend a_1 to be evaluated before a_0:

$$(3') \quad \frac{\langle a_1, \sigma\rangle \Downarrow n_1 \qquad \langle a_0, \sigma\rangle \Downarrow n_0}{\langle a_0 \texttt{ - } a_1, \sigma\rangle \Downarrow n} \quad n = n_0 - n_1$$

However, this gives rise to exactly the same evaluation relation, so an implementer of **IMP** would be free to choose a different evaluation order from the one suggested by the definition.

In **IMP**^{e}, however, a real difference emerges between rules (3) and (3'): there are programs that yield a different result depending on which rule we specify. [Exercise: give an example!] A formal semantics for the language will typically capture aspects of evaluation order that are significant for the behaviour of programs, and will thus specify exactly how such programs are intended to behave.

As a related example, we could have given the rules for and in a different way: In place of rules (9) and (10), we could have given the single rule

$$(9') \quad \frac{\langle b_0, \sigma\rangle \Downarrow t_0 \qquad \langle b_1, \sigma\rangle \Downarrow t_1}{\langle b_0 \text{ and } b_1, \sigma\rangle \Downarrow t} \quad t = t_0 \wedge t_1$$

where the operation $\wedge$ is defined by means of the usual truth-table. What difference would this make? In contrast to the old definition, this new definition suggests that even if b_0 evaluates to *false*, we need to plough on and evaluate b_1. In fact, in the case of **IMP**, it will not make any difference to the semantics if we do this (it will merely make the execution inefficient). In the case of **IMP**^{e}, however, there is a genuine difference between the two kinds of and. [Exercise: give a program that illustrates this.] Note that in Java, both kinds of and are provided as primitives: && is used for the "short circuit" version (also called "conditional and"), and & for the one that evaluates both arguments (also called "strict and").

John Longley