Formal Programming Language Semantics note 4

Semantics of IMP: the fine details

This note gives the complete set of semantic rules of the language **IMP**, and goes into some of the finer points concerning this style of semantics.

Rules for arithmetic expressions

(1) $$\frac{}{\langle n, \sigma\rangle \Downarrow n}$$

(2) $$\frac{}{\langle X, \sigma\rangle \Downarrow \sigma(X)}$$

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

(4) Similar to (3) but for `*` .

Rules for boolean expressions

(5) $$\frac{}{\langle t, \sigma\rangle \Downarrow t}$$

(6) $$\frac{\langle a_0, \sigma\rangle \Downarrow n_0 \qquad \langle a_1, \sigma\rangle \Downarrow n_1}{\langle a_0 \texttt{ <= } a_1, \sigma\rangle \Downarrow \mathit{true}} \quad n_0 \leq n_1$$

(7) $$\frac{\langle a_0, \sigma\rangle \Downarrow n_0 \qquad \langle a_1, \sigma\rangle \Downarrow n_1}{\langle a_0 \texttt{ <= } a_1, \sigma\rangle \Downarrow \mathit{false}} \quad n_0 > n_1$$

(8) $$\frac{\langle b, \sigma\rangle \Downarrow t}{\langle \texttt{not } b, \sigma\rangle \Downarrow t'} \quad t' = \neg t$$

(9) $$\frac{\langle b_0, \sigma\rangle \Downarrow \mathit{false}}{\langle b_0 \texttt{ and } b_1\rangle \Downarrow \mathit{false}}$$

(10) $$\frac{\langle b_0, \sigma\rangle \Downarrow \mathit{true} \qquad \langle b_1, \sigma\rangle \Downarrow t}{\langle b_0 \texttt{ and } b_1\rangle \Downarrow t}$$

Rules for commands In rule (12) below we need a little bit of extra notation involving states. If $\sigma \in \mathbb{S}$, $X \in \mathbb{I}$ and $n \in \mathbb{Z}$, we write $\sigma[X \mapsto n]$ (informally "σ with X updated to n") for the state σ' defined by

$$\sigma'(Y) \;=\; \begin{cases} n & \text{if } Y = X, \\ \sigma(Y) & \text{otherwise.} \end{cases}$$

(11)
$$\frac{}{\langle \texttt{skip}, \sigma\rangle \Downarrow \sigma}$$

(12)
$$\frac{\langle a, \sigma\rangle \Downarrow n}{\langle X \texttt{ := } a, \sigma\rangle \Downarrow \sigma'} \quad \sigma' = \sigma[X \mapsto n]$$

(13)
$$\frac{\langle c_0, \sigma\rangle \Downarrow \sigma' \qquad \langle c_1, \sigma'\rangle \Downarrow \sigma''}{\langle c_0 \texttt{ ; } c_1, \sigma\rangle \Downarrow \sigma''}$$

(14)
$$\frac{\langle b, \sigma\rangle \Downarrow \mathit{true} \qquad \langle c_0, \sigma\rangle \Downarrow \sigma'}{\langle \texttt{if } b \texttt{ then } c_0 \texttt{ else } c_1, \sigma\rangle \Downarrow \sigma'}$$

(15)
$$\frac{\langle b, \sigma\rangle \Downarrow \mathit{false} \qquad \langle c_1, \sigma\rangle \Downarrow \sigma'}{\langle \texttt{if } b \texttt{ then } c_0 \texttt{ else } c_1, \sigma\rangle \Downarrow \sigma'}$$

(16)
$$\frac{\langle b, \sigma\rangle \Downarrow \mathit{false}}{\langle \texttt{while } b \texttt{ do } c, \sigma\rangle \Downarrow \sigma}$$

(17)
$$\frac{\langle b, \sigma\rangle \Downarrow \mathit{true} \qquad \langle c, \sigma\rangle \Downarrow \sigma' \qquad \langle \texttt{while } b \texttt{ do } c, \sigma'\rangle \Downarrow \sigma''}{\langle \texttt{while } b \texttt{ do } c, \sigma\rangle \Downarrow \sigma''}$$

We now discuss some aspects of this style of definition in more detail, making a few points somewhat more precise.

Rules vs. rule instances Let us use the term *evaluation statement* for a formula $\langle P, \sigma\rangle \Downarrow R$ (whether derivable or not), where P is a particular program phrase, σ is a particular state and R is a particular result of the appropriate type. A derivation tree thus consists of evaluation statements related by rule instances. Clearly, we may identify the set of all possible evaluation statements with the set

$$V \;=\; (\mathsf{Aexp} \times \mathbb{S} \times \mathbb{Z}) \sqcup (\mathsf{Bexp} \times \mathbb{S} \times \mathbb{T}) \sqcup (\mathsf{Com} \times \mathbb{S} \times \mathbb{S}).$$

Strictly speaking, the formulae occurring in the rules are not themselves evaluation statements, since they contain metavariables such as a, b, c, X, n, σ which range over various kinds of entities. Rather, they are patterns or "templates" which give rise to an infinite number of evaluation statements as particular *instances*. Thus, the rules themselves are really templates giving rise to an infinite set of *rule instances*, and it is these rule instances that occur in derivations. To

instantiate a rule, we must assign to each metavariable occurring in the rule an entity of the appropriate kind, in such a way that any side-conditions of the rule are satisfied. We must then replace each metavariable occurrence by the corresponding entity; Note that for each metavariable, all occurrences must be replaced by the same entity, whether they appear in the premises or conclusion of the rule.

We will write $\mathcal{I}$ for the set of rule instances obtained in this way from the rules of **IMP**.

Inductive definitions via rules Now that we have made precise the way in which a bunch of rules gives rise to a set of rule instances, let us be a little more explicit about how this gives rise to an evaluation relation. Recall that we are trying to define a subset E of the set V defined above. One possible definition of E is as follows:

$E = \{(P, \sigma, R) \mid \text{There's a derivation of } \langle P, \sigma\rangle \Downarrow R \text{ by means of rule instances in } \mathcal{I}\}$

To make this absolutely rigorous, one would have to provide some precise definition of a *derivation* as a mathematical object (easy but boring). However, there is another way of formulating the definition of E which is mathematically more appealing. It is worth making some effort to understand this, as it is an example of an *inductive definition* of a kind which arises very frequently in mathematics and computer science, especially language theory.

Let us say a subset $F \subseteq V$ is *closed under* $\mathcal{I}$ if whenever

$$\frac{\phi_1 \quad \cdots \quad \phi_k}{\phi}$$

is a rule instance in $\mathcal{I}$ and $\phi_1, \ldots, \phi_k \in F$, we also have $\phi \in F$. Clearly our set E should be closed under $\mathcal{I}$, since if $\phi_1, \ldots, \phi_k$ are all derivable and the above rule instance is in $\mathcal{I}$, then ϕ is also derivable. However, we also want to say that E doesn't contain any other evaluation statements beyond those that are entailed by the rules — in other words, E contains nothing more than is implied by saying it is closed under $\mathcal{I}$. We can make this precise as follows:

> Let E be the intersection of all subsets $F \subseteq V$ that are closed under $\mathcal{I}$.
> In symbols: $E = \bigcap\{F \subseteq V \mid F \text{ closed under } \mathcal{I}\}$.

Exercise: Show that E (defined in this way) is itself closed under $\mathcal{I}$, and is therefore the *smallest* subset of V that is closed under $\mathcal{I}$. Check that this definition of E agrees with the definition via derivations.

Whichever definition we adopt, it is clear that we can prove properties of the evaluation relation by induction. Specifically, to show that some property holds for all instances $\langle P, \sigma\rangle \Downarrow R$ of the evaluation relation, it is enough to show that this property is preserved by all the semantic rules.

Proving facts about programs It is possible in principle (though hard in practice!) to prove program correctness results directly from the operational definition of the language. For example, let P be the **IMP** program

```
while 1<=X do (Y:=Y*X ; X:=X-1)
```

which we considered at the end of Note 3. You might like to try to prove the following fact directly from the operational semantics:

$$\text{If } \sigma(\mathtt{X}) = m \text{ and } \sigma(\mathtt{Y}) = 1, \text{ then } \langle P, \sigma\rangle \Downarrow \sigma' \text{ where } \sigma'(\mathtt{X}) = 0 \text{ and } \sigma'(\mathtt{Y}) = m!.$$

This way of proving properties of programs is not recommended in general — as we shall see, the use of an axiomatic semantics is usually more convenient.

As well as properties of particular *programs*, we can also prove interesting facts about the *language*. For instance, in the case of **IMP**, one can prove the following important facts as theorems:

- For all a, σ there is a *unique* n such that $\langle a, \sigma\rangle \Downarrow n$.
- For all b, σ there is a unique t such that $\langle b, \sigma\rangle \Downarrow t$.
- For all c, σ there is *at most one* σ' such that $\langle c, \sigma\rangle \Downarrow \sigma'$.

[Exercise: satisfy yourself that you know how to prove these.] These facts tell us that the run-time behaviour of **IMP** is deterministic and completely specified by the rules we have given. In general, we say an evaluation relation is *deterministic* (or sometimes *monogenic*) if for every P, σ there is at most one R such that $\langle P, \sigma\rangle \Downarrow R$. Usually we will want our evaluation relation to be deterministic. However, we might sometimes want to consider non-deterministic relations, either because of some kind of run-time non-determinism in the language, or because we want to leave certain aspects of program behaviour unspecified in the language definition.

Aside: Big-step vs. small-step relations. The kind of relation we have been considering here is sometimes called a *big-step* evaluation relation, since it captures the idea of evaluating or executing a piece of code all the way to its final result. This contrasts with *small-step* evaluation relations, which typically capture the idea of a single "computation step". A small-step relation might have the form

$$\langle P, \sigma\rangle \rightarrow \langle P', \sigma'\rangle$$

and we can then obtain a complete computation by chaining together instances of this relation until we reach something that cannot be evaluated any further. A structural operational semantics may be given in either a big-step or a small-step style; big-step presentations tend to involve fewer arbitrary choices, while small-step presentations are easier to implement on a machine.

John Longley