Formal Programming Language Semantics note 3

Operational semantics of an imperative language

Let's start by giving the operational semantics of a simple imperative language, which we shall call **IMP**. This language is taken from Glynn Winskel's book — we will follow his presentation closely with some minor changes of notation. It is very similar to the language **LC** discussed in the CS3 Language Semantics and Implementation course.

Notation. We will use teletype font `like this` for sequences of symbols that may appear as part of a program text. We will write:

- $\mathbb{Z}$ for the set of integers $\ldots, -2, -1, 0, 1, 2, \ldots$. We use m, m', n, n' as variables ranging over $\mathbb{Z}$. For convenience, we will not bother to distinguish between mathematical integers like 5 and the corresponding *integer constants* like `5`, the latter being a symbolic representation that might appear in a program. (Philosophical conundrum: what ultimately *is* the number 5 anyway??)
- $\mathbb{T}$ for the set of truth values *true*, *false*. We use t, t' to range over $\mathbb{T}$. Again, we will identify truth values with the *boolean constants* `true`, `false`.
- $\mathbb{I}$ for the set of *identifiers*. An identifier is a string of characters (such as `X` or `Catch22`) suitable for use as a variable name in a program. We use X, X' to range over $\mathbb{I}$.

A complete description of a language at the *lexical* level would include a precise definition of which strings of symbols were valid identifiers, but we will not fuss over this here. The idea (for the moment) is that an identifier such as `X` can be used to refer to the location in memory where the value of the program variable `X` is stored. Notice the notational difference between `X`, which is an example of a particular identifier or location, and X, which we use as a variable ranging over all possible identifiers. To avoid confusion we will sometimes refer to things like `X` as *program variables*, and things like X as *metavariables*.

Syntax of IMP. The three phrase categories of **IMP** are:

- Aexp (arithmetic expressions), ranged over by $a, a_0, a_1, \ldots$.
- Bexp (boolean expressions), ranged over by $b, b_0, b_1, \ldots$.
- Com (commands), ranged over by $c, c_0, c_1, \ldots$.

We define the syntax of **IMP** by means of the following context-free grammar:

Aexp : $a ::= n \mid X \mid a_0$ `-` $a_1 \mid a_0$ `*` a_1

Bexp : $b ::= t \mid a_0$ `<=` $a_1 \mid$ `not` $b \mid b_0$ `and` b_1

Com : $c ::=$ `skip` $\mid X$ `:=` $a \mid c_0$ `;` $c_1 \mid$ `if` b `then` c_0 `else` $c_1 \mid$ `while` b `do` c

For simplicity, we have gone for a rather minimalist language here: for instance, we have dispensed with the forms a_0 `+` a_1, a_0 `=` a_1, b_0 `or` b_1, since these may be replaced respectively by the forms

$$a_0 \texttt{ - } (0 \texttt{ - } a_1), \qquad (a_0 \texttt{ <= } a_1) \texttt{ and } (a_1 \texttt{ <= } a_0), \qquad \texttt{not } ((\texttt{not } b_0) \texttt{ and } (\texttt{not } b_1)).$$

Notice that this grammar is ambiguous in several different ways: for instance, there are two possible parse trees for `while` b `do` c_0 `;` c_1. [Exercise: in what other ways is the above grammar ambiguous?] In a formal language definition, it is usually important to give an unambiguous grammar, so that we know e.g. which interpretation of the above phrase is the correct one. In general, there are well-known techniques for turning a context-free grammar into an equivalent unambiguous one (these are covered e.g. in the Language Processing thread of CS2), and if need be we can also modify the language to allow the use of brackets to specify the intended structure. However, all this tends to clutter the definition of the language. Since we know that we could do this if we had to, and since this is a course on semantics than syntax, we shall gloss over these issues and content ourselves with a slightly informal approach, freely adding brackets where needed to clarify the syntax of particular program phrases.

The notion of state. When we execute a program in a language like **IMP** on a machine, at any point in time the machine will be in a certain state, in which (for instance) the memory locations corresponding to program variables will hold certain values. In our mathematical model, let us say a *state* is simply a total function σ from $\mathbb{I}$ to $\mathbb{Z}$, mapping each identifier X to the value $\sigma(X)$ stored in location X. (Note that in **IMP** all program variables are of integer type.) We will write $\mathbb{S}$ for the set of all possible states and use $\sigma, \sigma', \sigma'', \ldots$ to range over $\mathbb{S}$.

Typically, the actual state of a real machine will involve a lot more information than is given by an element of $\mathbb{S}$: for instance, there will be the details of how program variables like `X` are mapped to actual memory addresses like `007F86D2`, and also probably some kind of stack which records which parts of the program still have to be executed. The point, though, is that functions $\sigma : \mathbb{I} \to \mathbb{Z}$ give us an *abstract* notion of state which contains all the information we need in order to define how programs in **IMP** should behave. Thus, we can specify the semantics of **IMP** whilst freeing ourselves from most of the grungy details of the machine state (which, in any case, ought to be left up to the implementer).

At the moment, our notion of state will be a bit unrealistic if the set $\mathbb{I}$ is infinite (or even of size 10^{30}), since a real machine state will not contain an allocated

memory cell for every possible identifier. For the time being, then, let us suppose that $\mathbb{I}$ is finite and quite small, and that we are only considering the behaviour of programs involving a fixed set of variables for which memory has already been allocated and some value assigned. Later on, we will consider more realistic languages where new variables may be declared in the course of a program.

The evaluation relation. Now we come to the key idea of structural operational semantics. We will define an *evaluation relation* for the language **IMP**, which tells us what is supposed to happen when we evaluate or execute a certain program phrase starting in a certain state. The evaluation relation will have the form

$$\langle P, \sigma \rangle \Downarrow R$$

which informally will mean

> "If the phrase P is evaluated or executed starting in the state σ, the resulting computation terminates and yields the result R."

Here P is a phrase of any of our three categories Aexp, Bexp, Com; σ is an element of $\mathbb{S}$; and what R is will depend on the phrase category of P. That is, the appropriate notion of "result" will be different for each phrase category:

- The result of evaluating an arithmetic expression will be an integer, so the corresponding instances of the evaluation relation have the form $\langle a, \sigma \rangle \Downarrow n$.
- The result of evaluating a boolean expression will be a truth value, so the corresponding instances of the evaluation relation have the form $\langle b, \sigma \rangle \Downarrow t$.
- The result of evaluating a command will be a state (informally, the new state after the command has been executed), so the corresponding instances of the evaluation relation have the form $\langle c, \sigma \rangle \Downarrow \sigma'$.

Slightly more formally, our evaluation relation will be a certain subset

$$E \subseteq (\mathsf{Aexp} \times \mathbb{S} \times \mathbb{Z}) \sqcup (\mathsf{Bexp} \times \mathbb{S} \times \mathbb{T}) \sqcup (\mathsf{Com} \times \mathbb{S} \times \mathbb{S}),$$

where $\times$ means product of sets and $\sqcup$ means *disjoint union*. We can then regard $\langle P, \sigma \rangle \Downarrow R$ as a more readable expression for $(P, \sigma, R) \in E$.

The evaluation relation is defined by means of a *proof system*. that is, we give a set of *semantic rules* for deriving the true statements of the form $\langle P, \sigma \rangle \Downarrow R$. The complete set of semantic rules for **IMP** is given in the accompanying Note 4.

Notice that each rule has the form

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

with some number of *premises* written above the line (possibly zero!), and a single *conclusion* written below the line. Some rules also involve *side-conditions* written to the right of the rule — these record any restrictions on when the rule is supposed to apply. The intended meaning of a rule of the above form is:

> "If $\langle P_1, \sigma_1\rangle \Downarrow R_1$ and ... and $\langle P_k, \sigma_k\rangle \Downarrow R_k$ (and any side-conditions are also satisfied), then $\langle P, \sigma\rangle \Downarrow R$."

Note that in the case $k = 0$, the "if" condition here is vacuous, and so in this case the rule will simply say that $\langle P, \sigma\rangle \Downarrow R$.

Given this set of rules, we can plug together valid instances of them (that is, instances that satisfy any side-conditions) to form proof trees or *derivations* in an obvious way. Thus, a derivation of $\langle P, \sigma\rangle \Downarrow R$ will be a finite tree with this formula at the root, such that every formula in the tree follows from the formulae immediately above it via one of the semantic rules. (Note that the leaves of such a tree will necessarily be instances of rules with no premises — these play the role of "axioms".) As a simple example, below is a derivation for the execution of the command `if X<=0 then X:=0-X else skip` in a state σ where $\sigma(\mathtt{X}) = -5$. We write σ' for the updated state $\sigma[\mathtt{X} \mapsto 5]$.

$$\dfrac{\dfrac{\dfrac{}{\langle \mathtt{X}, \sigma\rangle \Downarrow -5} \quad \dfrac{}{\langle \mathtt{0}, \sigma\rangle \Downarrow 0}}{\langle \mathtt{X{<}{=}0}, \sigma\rangle \Downarrow \mathit{true}} \quad \dfrac{\dfrac{}{\langle \mathtt{0}, \sigma\rangle \Downarrow 0} \quad \dfrac{}{\langle \mathtt{X}, \sigma\rangle \Downarrow -5}}{\langle \mathtt{0\text{-}X}, \sigma\rangle \Downarrow 5}}{\langle \mathtt{X:=0\text{-}X}, \sigma\rangle \Downarrow \sigma'}}{\langle \mathtt{if\ X{<}{=}0\ then\ X:=0\text{-}X\ else\ skip}, \sigma\rangle \Downarrow \sigma'}$$

We may now (at long last) *define* the evaluation relation of **IMP** to be the set of triples (P, σ, R) such that $\langle P, \sigma\rangle \Downarrow R$ has a derivation of this kind. This completes the definition of the operational semantics of **IMP**.

Note that the rules are *syntax-directed* — that is, each rule is associated with a particular syntactic construct of **IMP**, represented by the phrase P occurring in the conclusion of the rule; moreover, the phrases occurring in the premises are syntactic components of P. (Indeed, they are strictly smaller than P in all except rule (17).) This means that, given a phrase P and a state σ, we may try to build a derivation for some $\langle P, \sigma\rangle \Downarrow R$ (discovering what R is in the process) by a simple depth-first search. If some such derivation exists, our search will find it after a finite time. You will see what I mean if you try out an example by hand:

Recommended exercise. Construct a complete derivation for the execution of the command `while 1<=X do (Y:=Y*X ; X:=X-1)`, starting from a state σ in which $\sigma(\mathtt{X}) = 3$, $\sigma(\mathtt{Y}) = 1$.

The construction of derivations in this way by traversing the proof from left to right in some sense simulates the *execution* of programs in **IMP**. Doing this by hand isn't very feasible except for tiny examples, but it does lend itself to machine simulations: we can provide the simulator with a set of operational rules, and instantly get a simulation of the corresponding programming language.

[Exercise: Think about what will happen when we attempt to construct a derivation in this way for a program that doesn't terminate.]

John Longley