Formal Programming Language Semantics note 6

A language with side-effects

We now consider another extension of **IMP** which embodies a typical characteristic of real programming languages: namely, that the evaluation of expressions not only yields a value but can also have side-effects, such as modifying the state or printing some output. In real languages, this can happen in numerous ways — for instance, an expression might consist of a function call, and the execution of the function body might print something or change the value of some variable — but here we will illustrate the principle by considering the *postincrement* operator familiar from C and Java.

A language with postincrement Let **IMP**^{s} be the language obtained by extending the syntax of **IMP** with a new phrase form for arithmetic expressions:

$$a \ ::= \ X\texttt{++}$$

Informally, in any state, an expression X++ will evaluate to the same value as X, but in the case of X++ the evaluation will have the *side-effect* of incrementing X. (Note that the evaluation yields the old value of X, not the new one. In C and Java there is also an analogous *preincrement* operator which yields the new value.)

To give a semantics for **IMP**^{s}, we will clearly need to capture not only what expressions evaluate to but also their effect on the state. (Note that boolean expressions can also have side-effects, since they may contain arithmetic expressions). We should therefore change the forms of evaluation statements for arithmetic and boolean expressions to

$$\langle a, \sigma\rangle \Downarrow \langle n, \sigma'\rangle, \qquad \langle b, \sigma\rangle \Downarrow \langle t, \sigma'\rangle.$$

The appropriate semantic rule for postincrement should obviously be

$$(18s) \quad \frac{}{\langle X\texttt{++}, \sigma\rangle \Downarrow \langle n, \sigma'\rangle} \ \ \sigma(X) = n, \sigma' = \sigma[X \mapsto n+1]$$

The other rules that explicitly deal with state — rules (2) and (12) of Note 4 — should now be changed to

$$(2s) \quad \frac{}{\langle X, \sigma\rangle \Downarrow \langle n, \sigma\rangle} \ \ \sigma(X) = n$$

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

None of the other rules in the definition of **IMP** explicitly manipulates the state. However, we will also need to change many of the other rules to accommodate the fact that the forms of evaluation statements have changed. It is obvious what changes we need to make: for instance, rules (3) and (17) should now become

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

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

There is an obvious pattern here: the result state occurring in each premise matches up, domino-fashion, with the initial state occurring in the following premise. This simply captures the natural result of evaluating the various constituents of the phrase in the appropriate order. In fact, one can codify this pattern as a general rule, known as the *state convention*. Like the exception convention mentioned in the previous note, this saves us a lot of tedious rewriting of rules and helps to keep the clutter to a minimum. (The state convention was used to good effect in this way in the definition of ML.)

To formulate the state convention precisely, it will actually be helpful to start by *erasing* the state information from the rules of **IMP** (except for rules (2) and (12), which explicitly make use of the state). Thus, in the remaining rules, we may strip the three forms of the evaluation relation down to $a \Downarrow n$, $b \Downarrow t$, and $c \Downarrow$ respectively. Let us call these formulae *reduced evaluation statements*. This allows for a more concise presentation of the rules: for example, rules (3) and (17) become respectively:

$$\frac{a_0 \Downarrow n_0 \qquad a_1 \Downarrow n_1}{a_0 \texttt{ - } a_1 \Downarrow n} \quad n = n_0 - n_1 \qquad\qquad \frac{b \Downarrow \mathit{true} \qquad c \Downarrow \qquad \texttt{while } b \texttt{ do } c \Downarrow}{\texttt{while } b \texttt{ do } c \Downarrow}$$

Given a reduced evaluation statement ϕ and two state metavariables σ, σ', let us write $\sigma\sharp\phi\sharp\sigma'$ for the reconstituted evaluation statement defined as follows:

$$\begin{aligned} \sigma\sharp(a \Downarrow n)\sharp\sigma' &\equiv \langle a, \sigma\rangle \Downarrow \langle n, \sigma'\rangle, \\ \sigma\sharp(b \Downarrow t)\sharp\sigma' &\equiv \langle b, \sigma\rangle \Downarrow \langle t, \sigma'\rangle, \\ \sigma\sharp(c \Downarrow)\sharp\sigma' &\equiv \langle c, \sigma\rangle \Downarrow \sigma' \end{aligned}$$

We may now make precise the state convention as follows: given a formal rule

$$\frac{\phi_1 \quad \phi_2 \quad \cdots \quad \phi_k}{\phi} \text{ side-cond}$$

where the ϕ_i and ϕ are reduced evaluation statements, we understand this to abbreviate the ordinary rule:

$$\frac{\sigma_0\sharp\phi_1\sharp\sigma_1 \qquad \sigma_1\sharp\phi_2\sharp\sigma_2 \qquad \cdots \qquad \sigma_{k-1}\sharp\phi_k\sharp\sigma_k}{\sigma_0\sharp\phi\sharp\sigma_k} \text{ side-cond}$$

(where the side-condition is the same as in the reduced rule).

To summarize, the complete semantic rules for **IMP**^{s} are: the erased forms of rules (1), (3)–(11) and (13)–(17) from Note 4 (with the state convention applied), together with rules (2s), (12s) and (18s) above.

Languages with output The idea behind the state convention can usefully be applied to describe other kinds of side-effects. For example, let **IMP**^{o} be the language **IMP** extended with the arithmetic expression form `print` a, which evaluates to the value of a but with the side-effect of outputting the value of a. Thus, the output obtained from a program will in general be a list of integers.

To give a semantics for this language, let us write $\mathbb{O}$ for the set of lists of integers, with $o, o_1, \ldots$ ranging over $\mathbb{O}$. We write $o; n$ for extension of a list by a single element, and $o_1 @ o_2$ for concatenation of lists. In place of the evaluation relation $\langle P, \sigma\rangle \Downarrow R$ of **IMP**, we now define an relation $\langle P, \sigma\rangle \Downarrow_o R$, meaning "in state σ, running P yields the result R and the output o". The rule for `print` is

$$(19s) \quad \frac{\langle a, \sigma\rangle \Downarrow_o n}{\langle \texttt{print}\ a, \sigma\rangle \Downarrow_{o'} n} \quad o' = o; n$$

The remaining rules for **IMP**^{o} can then be obtained from the (original) rules for **IMP** by means of the following *output convention:*

$$\frac{\langle P_1, \sigma_1\rangle \Downarrow R_1 \cdots \langle P_k, \sigma_k\rangle \Downarrow R_k}{\langle P, \sigma\rangle \Downarrow R} \text{ becomes } \frac{\langle P_1, \sigma_1\rangle \Downarrow_{o_1} R_1 \cdots \langle P_k, \sigma_k\rangle \Downarrow_{o_k} R_k}{\langle P, \sigma\rangle \Downarrow_o R}\ o = o_1 @ \ldots @ o_k$$

(Side-conditions in the original rule must also be included in the new rule.)

[Exercises: Add some further machinery to specify the output produced by non-terminating programs. Think about how one might describe a language with input as well as output.]

Patterns and observational equivalence An important question, both for programmers and compiler writers, is "When do two program phrases P_1, P_2 yield the same behaviour?". If they do, we (or a compiler) can legitimately replace the less efficient one (P_1 say) by the more efficient one (P_2), knowing that the overall result of the program will be unaffected.

A very important point, which we wish to emphasize in this course, is that the answer to the above question depends very much on the language in question. For instance, from the discussion at the end of Note 5, it should be intuitively clear that in **IMP**, the phrases b_0 `and` b_1 and b_1 `and` b_0 have the same behaviour for any boolean expressions b_0, b_1, while this is not true in **IMP**^{e}, nor indeed in **IMP**^{s}, since evaluation order clearly matters when we have side effects. [Exercise: give an example to show this.] Likewise, in both **IMP** and **IMP**^{e} the phrases b and b `and` b always have the same behaviour (satisfy yourself that this is true), though in **IMP**^{s} they do not (give an example to show this). In general, one can say that the larger the programming language, the more "fine-grained" this kind of behavioural equivalence will be.

Using the operational ideas we have seen so far, we can formulate some precise definitions to capture the intuitive notion of "having the same behaviour". The question of how to *prove* that two given phrases have the same behaviour (if they do!) is something we shall return to later — it is one of the things that denotational semantics is good for.

In the following definitions, let L be any programming language equipped with an evaluation relation $\langle P, \sigma\rangle \Downarrow R$ of the kind discussed in the notes.

- A *pattern* of L is (loosely speaking) a phrase of L which may contain metavariables which each range over a particular phrase category. For instance, `if` a_0 `=` a_1 `then` c_0 `else` c_1 is a pattern of **IMP** containing the metavariables a_0, a_1 ranging over Aexp, and c_0, c_1 ranging over Com. [If you want to be more formal about it, one can obtain a grammar for patterns from a grammar for L by adding in a clause for metavariables of each phrase category, e.g. $a ::=$ *Aexp–metavar*.]

- A *substitution* γ for L is any function mapping metavariables to phrases of L of the appropriate category.

- Given a pattern P and a substitution γ which covers all the metavariables of P, we write P^γ for the phrase obtained from P by replacing each metavariable x by the corresponding phrase $\gamma(x)$.

- Let us say two patterns P_1, P_2 are *functionally equivalent* in L (this is not quite standard terminology!) if for any substitution γ for L covering all the metavariables of P_1, P_2, any state σ and any result R we have

$$\langle P_1^\gamma, \sigma\rangle \Downarrow R \quad \text{iff} \quad \langle P_2^\gamma, \sigma\rangle \Downarrow R.$$

- A *context* of L is a pattern involving a single metavariable (which may occur more than once in the pattern). If C is a context, we write $C[P]$ for the phrase obtained from by replacing each occurrence of the metavariable by P (assuming P is of the right phrase category).

- We say two patterns P_1, P_2 are *observationally equivalent* in L if for any context C of the appropriate type, and any substitution γ, state σ and result R as above, we have

$$\langle C[P_1^\gamma, \sigma]\rangle \Downarrow R \quad \text{iff} \quad \langle C[P_2^\gamma, \sigma]\rangle \Downarrow R.$$

Thus, to revisit the examples mentioned above, the patterns b_0 `and` b_1, b_1 `and` b_0 are observationally equivalent in **IMP**, though not in **IMP**^{e} or **IMP**^{s}, while the patterns b, b `and` b are observationally equivalent in **IMP** and **IMP**^{e} but not **IMP**^{s}.

[Once you have digested all this, here are some further questions you might like to ponder: (a) Is observational equivalence actually stronger than functional equivalence for the languages we've considered? (b) Can two patterns be observationally equivalent in **IMP**^{s} but not in **IMP**^{e}? (c) How does observational equivalence in **IMP**^{o} compare with the others?]

John Longley