AN OPERATIONAL SEMANTICS FOR CSP

(Extended Abstract)

Gordon Plotkin

The present work is intended to illustrate a method of giving the operational semantics of programming languages. As an example a variant of Hoare's language, CSP, of communicating sequential processes is considered [Hoa]. At the same time some attempt is made to develop an approach to language analysis following a guiding principle of simplicity of the abstract syntax. This idea is independent of the semantic method used and can just as well be combined with other approaches, such as that of denotational semantics.

The operational semantics employs the well-known idea of transition systems, $\langle \Gamma, T, \rightarrow \rangle$, where Γ (ranged over by γ) is a set of configurations and $T \subseteq \Gamma$ is the set of final configurations and $\rightarrow \subseteq \Gamma \times \Gamma$ is the transition relation. A typical configuration could be $\langle c, \sigma \rangle$, where c is a command and σ is a store; further any store σ could be a final configuration (no command left to be executed). What is new is the specification of the transition relation. For this we follow the modern emphasis on structure and define the transition relation by structural induction on the abstract syntax by means of such rules as:

Sequence 1. $$\frac{\langle c_0, \sigma \rangle \rightarrow \langle c_0', \sigma' \rangle}{\langle c_0; c_1, \sigma \rangle \rightarrow \langle c_0'; c_1, \sigma' \rangle}$$

2. $$\frac{\langle c_0, \sigma \rangle \rightarrow \sigma'}{\langle c_0; c_1, \sigma \rangle \rightarrow \langle c_1, \sigma' \rangle}$$

Dijkstra's guarded command language underlies CSP [Dij]. Applying our principle of simplicity we try to minimise the number of syntactic categories in a non-artificial way and take guarded commands as being themselves commands so that for example if b is a Boolean expression and c is a command so is $b \rightarrow c$. As another application we do not wish to allow n-ary syntactic constructions (for every n) as implied by this syntax for conditions:

$$\underline{\text{if}}\ b_1 \rightarrow c_1 \ [] \ \ldots \ [] b_n \rightarrow c_n\ \underline{\text{fi}}$$

Rather we regard this, for each n, as a composite (= derived) construction from the above binary guarding construction, $b \rightarrow c$, a binary choice construction $c \ [] \ c$ and a unary conditional if c fi. The syntax for commands is then

c::= a | skip | c;c | b => c | fail | c [] c |
abort | if c fi | do c od

where a ranges over primitive actions and b => c is a strong guarding construction, useful for concurrency, which does not allow any interruption by any process between the testing of b and the first action of c. The above weak guarding can be derived from the strong one by:

$$b \rightarrow c =_{def} b \Rightarrow nil; c$$

This can be expressed by the rules:

Guarding 1. $\dfrac{\langle c, \sigma\rangle \rightarrow \gamma}{\langle b \Rightarrow c, \sigma\rangle \rightarrow \gamma}$ (if b is true in σ)

2. $\langle b \Rightarrow c, \sigma\rangle \rightarrow$ failure (if b is false in σ)

where failure is a new configuration standing for failure. The other constructs are specified along similar lines (and another new configuration, abortion, is needed).

The advantage of the strong guarding is that it allows the resolution of complex guarded commands in CSP containing input commands, such as b; A ? x → c, into b => (A ? x; c) and that in turn avoids the redundancy of input commands appearing in two places in the syntax.

The most complex construct in CSP is the parallel command

$$[P_1::c_1 \,||\, \ldots \,||\, P_n::c_n]$$

where, of course, the c_i can perfectly well contain further parallel commands. We view this as a block (with || replacing ;) and regard the process identifiers, P_i, as analogous to label identifiers; then the scope of the P_i can be taken, following the usual rules, as the whole of the block. With this idea we can analyse the parallel command into a binary parallel construction, c||c and a unary labelling construction P::c and a binary declaration construction process P;c as follows

process P_1; ...; process P_n;
begin
$P_1::c_1 \,||\, \ldots \,||\, P_n::c_n$
end

(where begin ... end are just a pair of brackets). And we have a simple syntax for CSP

c::= (guarded command construction) | P?r | Q!w |
c || c | P::c | process P; c

where r ranges over a set of input operations and w over a set of output operations and we have omitted certain details relating to non-interference between parallel processes. We have also omitted variable declarations, process arrays or guarded commands with ranges.

For the operational semantics we employ the well-known idea of labelled transition systems, $\langle \Gamma, T, M, \rightarrow \rangle$ where now M is a set of messages (= communications) and $\rightarrow \subseteq \Gamma \times M \times \Gamma$.. The rules for the guarded commands are the same as before, but with an m above the arrows for the messages. For CSP the appropriate messages with their intuitive meanings are:

- ε - an internal action of a process
- P ? v - P sends v to an unknown agent
- Q ! v - an unknown agent sends a value v to Q
- P,Q ? v - Q receives v from P
- P,Q ! v - P sends v to Q.

Here are some typical rules

Output $\langle Q \,!\, w, \sigma \rangle \xrightarrow{Q!v} \sigma$ (where v is the value transmitted by w given store σ)

Labelling $$\frac{\langle c, \sigma \rangle \xrightarrow{P?v} \langle c', \sigma' \rangle}{\langle Q::c, \sigma \rangle \xrightarrow{P,Q?v} \langle Q::c', \sigma' \rangle} \qquad (\text{if } P \neq Q)$$

Communication $$\frac{\langle c_0, \sigma \rangle \xrightarrow{P,Q?v} \langle c_0', \sigma' \rangle \quad \langle c_1, \sigma \rangle \xrightarrow{P,Q!v} \langle c_1', \sigma \rangle}{\langle c_0 || c_1, \sigma \rangle \xrightarrow{\varepsilon} \langle c_0' || c_1', \sigma' \rangle}$$

This treatment omits consideration of Hoare's termination convention, which would require a little additional complication. (One adds a third component to the configurations, namely a set of process labels; these are intended to be the set of labels of all terminated processes.) Finally we note a great difference between this kind of operational semantics and the usual denotational semantics. We only specify one step of executions and do not say how to put these together to make up global behaviours or what it means for two such behaviours to be equal. There are many choices in general and we remain neutral between them. On the other hand any normal denotational semantics both automatically reflects such a choice and does not in general specify all the details of the execution sequences.

References

[Dij] Dijkstra, E.W. (1976) A Discipline of Programming. Prentice-Hall.

[Hoa] Hoare, C.A.R. (1978) Communicating Sequential Processes. CACM, Vol. 21, No. 8, 666-677.