Formal Semantics of Programming Languages

Mooly Sagiv, Eran Yahav

Lecture #9, 8 May 2003: **Nondeterminism and Parallelism**
Notes by: Dana Fisman

In this lecture we introduce nondeterministic and parallel (or concurrent) programs and their semantics.

A simple parallel construct

The parallel composition of two commands c_0 and c_1 is denoted $c_0 \parallel c_1$. The intended meaning is that both commands c_0 and c_1 are executed with no particular order; and, moreover, their execution may be interleaved. For example, if c_0 is $X := 1$ and c_1 is $X := 5;\ X := X + 1$ then their parallel composition

$$(X := 1 \parallel (X := 5;\ X := X + 1))$$

can be executed is 3 ways:

- First c_0 is executed entirely and then c_1 is executed entirely. In this case the value of X at the end of the execution is 6.
- First c_1 is executed entirely and then c_0 is executed entirely. In this case the value of X at the end of the execution is 1.
- First the execution of c_1 begins. The command $X := 5$ is executed. Next the command $X := 1$ is executed (that is c_0 begins and ends to execute). Finally, the command $X := X + 1$ is executed (that is c_1 terminates to execute). The resulting value of X in this case is 2.

This example shows that parallel composition introduces unpredictability or *nondeterminism*[1]. In the example above, we cannot know in advance the value of X at the end of execution of $c_0 \parallel c_1$. And, thus, in general we cannot know the result of executing a parallel command.

An immediate consequence of the above is that we cannot model the execution of parallel commands by means of a relation between commands and *final* states. Thus, we cannot define parallel composition using natural operational semantics. This, since the result depends on intermediate values of the commands executed in parallel – details which are hidden in the natural semantics. Since intermediate results are available in the small-step semantics, we can use it to provide a semantics for parallel composition.

The following rules provide a small-step semantic for the simple parallel composition command.

[1] This nondeterminism is sometimes refereed to as *demonic* non-determinism since all alternatives, including "bad alternatives" have to be considered. This should be contrasted with *angelic* (also known as *mnemonic*) non-determinism which allows considering a single "good" alternative when such alternative exists.

$$[\mathbf{P}_1]\quad \frac{\langle c_0,\sigma\rangle \to_1 \sigma'}{\langle c_0 \parallel c_1,\sigma\rangle \to_1 \langle c_1,\sigma'\rangle} \qquad [\mathbf{P}_2]\quad \frac{\langle c_0,\sigma\rangle \to_1 \langle c_0',\sigma'\rangle}{\langle c_0 \parallel c_1,\sigma\rangle \to_1 \langle c_0' \parallel c_1,\sigma'\rangle}$$

$$[\mathbf{P}_3]\quad \frac{\langle c_1,\sigma\rangle \to_1 \sigma'}{\langle c_0 \parallel c_1,\sigma\rangle \to_1 \langle c_0,\sigma'\rangle} \qquad [\mathbf{P}_4]\quad \frac{\langle c_1,\sigma\rangle \to_1 \langle c_1',\sigma'\rangle}{\langle c_0 \parallel c_1,\sigma\rangle \to_1 \langle c_0 \parallel c_1',\sigma'\rangle}$$

Rules $\mathbf{P}_1$ and $\mathbf{P}_2$ take account of the case where the parallel composition $c_0 \parallel c_1$ starts by an execution of the first command in c_0. If the execution then completes, it remains to execute c_1; this is captured by rule $\mathbf{P}_1$. Otherwise, if c_0 did not complete its execution, then it remains to execute $c_0' \parallel c_1$ assuming c_0' is what remains to execute to complete c_0. The rules $\mathbf{P}_3$ and $\mathbf{P}_4$ capture the symmetric situation in which c_1 begins to execute before c_0. The symmetry shows that there is indeed no preference in executing (or more accurately, starting to execute) one command before the other. Since in the parallel execution $c_0 \parallel c_1$ the component commands may influence each other (if they have common locations), this can be thought of, and is often refereed to as *communication by shared variables*.

Non-determinism and Guarded commands

The non-determinism in the above example is artificial and perhaps gives the impression that it can be avoided. Non-determinism is however very useful. We will see in the sequel several examples of programs (which can be thought of as algorithms) where the use of non-determinism is very natural and makes the programs much more elegant than the equivalent deterministic programs.

For this purpose we introduce Dijkstra's language of guarded commands. Dijkstra's idea is that a nondeterministic construction can help free the programmer from overspecifying a method of solution. It is often the case that a task can be distributed to several small tasks each may be carried out provided some assumptions hold, and the big task is carried out when any of its sub-tasks is carried out. Dijkstra's guarded commands are extremely convenient to specify algorithms written in such a spirit. Hopefully, this idea will become clear after we'll see several examples.

Dijkstra's language consists of arithmetic and boolean expressions, commands and guarded commands. For the arithmetic and boolean expressions we use the sets **Aexp** and **Bexp** of **IMP**. The abstract syntax of the commands **DCom** and guarded commands **GC** is given below (we use a, b, c, and gc to range over **Aexp**, **Bexp**, **DCom** and **GC**, respectively).

$$c ::= \mathbf{skip} \mid \mathbf{abort} \mid X := a \mid c_0; c_1 \mid \mathbf{if}\ gc\ \mathbf{fi} \mid \mathbf{do}\ gc\ \mathbf{od}$$

$$gc ::= b \to c \mid gc_0\ [\!]\ gc_1$$

The commands **skip**, assignment and sequential composition have the same meaning in **IMP**. The new command **abort** causes the program to "crash" or get *stuck*. It does so by not yielding a final state from any initial state. An abort is different from an infinite loop (such as **while true do skip**); in an infinite loop the program goes through an infinite sequence of states as opposed to the abort command which crashes the system in one iteration.

The boolean expression b in the guarded command $b \to c$ is called a *guard*; the command body c is executed only if the guard b evaluates to true. Otherwise the guarded command

is said to *fail*. The constructor used to form the guarded command $gc_0 \,[]\, gc_1$ is called *alternative* or "fat-bar" (sometimes the box symbol $\Box$ or the word **or** are used instead). A guarded command typically takes the form

$$(b_1 \to c_1) \,[]\, (b_2 \to c_2) \,[]\, \cdots \,[]\, (b_n \to c_n)$$

Intuitively then, the execution of the guarded command results in the execution of one of its command bodies (c_i), the respective guard of (b_i) was evaluated to true. The execution is said to *fail* if none of the guards is evaluated to true.

The command **if** gc **fi** executes as the guarded command gc if gc does not fail, and executes like **abort** otherwise. The command **do** gc **od** repeatedly executes the guarded command gc as long as gc does not fail, and terminates once gc fails. Thus, if gc fails initially the command **do** gc **od** acts like **skip**.

We note that the lack of conditional or while statement such as in **IMP** does not reduce the strength of Dijkstra's language. The command (**if** b **then** c_0 **else** c_1) can be expressed in Dijkstra's language as

$$\textbf{if } (b \to c_0) \,[]\, (\neg b \to c_1) \textbf{ fi}$$

Similarly the command (**while** b **do** c) can be expressed by

$$\textbf{do } b \to c \textbf{ od}$$

We give two examples before we provide the formal semantics for the language.

Example.
The following programm computes (in location MAX) the maximum between two values (in locations X and Y):

$$\begin{array}{l} \textbf{if} \\ \quad X \geq Y \to MAX := X \\ \quad [] \\ \quad Y \geq X \to MAX := Y \\ \textbf{fi} \end{array}$$

Compare this with the following, more traditional program in **IMP**:

$$\textbf{if } X \geq Y \textbf{ then } MAX := X \textbf{ else } MAX := Y;$$

in which the symmetry between X and Y is broken. The asymmetry between the two branches of a conditional statement (in **IMP**) is the cause of the symmetry break.

Example.
The following programs computes Euclids algorithm for the greatest common divisor of two positive integers

$$
\begin{array}{l}
\textbf{do} \\
\quad X > Y \to X := X - Y \\
\quad [] \\
\quad Y > X \to Y := Y - X \\
\textbf{od}
\end{array}
$$

Again, we see that the symmetry is broken in the more traditional **IMP** program:

$$
\begin{array}{l}
\textbf{while } \neg(X = Y) \textbf{ do} \\
\quad \textbf{if } X \geq Y \textbf{ then } X := X - Y \textbf{ else } Y := Y - X;
\end{array}
$$

More examples can be found in Dijkstra's book [1].

We now turn to the formal semantics of Dijkstra's language. For arithmetic and boolean expression we assume the rules are given in natural semantics as in **IMP**. For the commands and guarded commands we provide the rules in a small-step semantics rather than in a natural semantics in order to enhance it later to include parallelism (and since, as discussed earlier, parallelism cannot be defined using natural semantics).

Rules for commands

$$[\mathbf{C}_1] \quad \langle \textbf{skip}, \sigma \rangle \to_1 \sigma$$

$$[\mathbf{C}_2] \quad \frac{\langle a, \sigma \rangle \to_1 n}{\langle X := a, \sigma \rangle \to_1 \sigma[n/X]}$$

$$[\mathbf{C}_3] \quad \frac{\langle c_0, \sigma \rangle \to_1 \sigma'}{\langle c_0; c_1, \sigma \rangle \to_1 \langle c_1, \sigma' \rangle} \qquad [\mathbf{C}_4] \quad \frac{\langle c_0, \sigma \rangle \to_1 \langle c_0', \sigma' \rangle}{\langle c_0; c_1, \sigma \rangle \to_1 \langle c_0'; c_1, \sigma' \rangle}$$

$$[\mathbf{C}_5] \quad \frac{\langle gc, \sigma \rangle \to_1 \langle c, \sigma' \rangle}{\langle \textbf{if } gc \textbf{ fi}, \sigma \rangle \to_1 \langle c, \sigma' \rangle}$$

$$[\mathbf{C}_6] \quad \frac{\langle gc, \sigma \rangle \to_1 \textbf{fail}}{\langle \textbf{do } gc \textbf{ od}, \sigma \rangle \to_1 \sigma} \qquad [\mathbf{C}_7] \quad \frac{\langle gc, \sigma \rangle \to_1 \langle c, \sigma' \rangle}{\langle \textbf{do } gc \textbf{ od}, \sigma \rangle \to_1 \langle c; \textbf{do } gc \textbf{ od}, \sigma' \rangle}$$

The absence of a rule for **abort** is intentional. By not having a rule for it we get the desired result that the execution gets *stuck* upon executing an abort command (since no final state can be derived). Alternatively we could make the abortion explicit by introducing a new command configuration implying improper termination. Similarly, since there is no rule for the command **if** gc **fi** when gc fails, the results of executing it when gc fails is the same as **abort**.

The guarded commands themselves do not cause side effects. Thus, the premise $\langle gc, \sigma \rangle \to_1 \langle c, \sigma' \rangle$ in rules $\mathbf{C}_5$ and $\mathbf{C}_7$ might as well be written as $\langle gc, \sigma \rangle \to_1 \langle c, \sigma \rangle$. We chose the previous option in order to ease a later extension of the language to cases where the guarded commands do cause side effects.

Rules for guarded commands

$$[\mathbf{GC}_1] \quad \frac{\langle b, \sigma\rangle \to_1 \mathbf{true}}{\langle b \to c, \sigma\rangle \to_1 \langle c, \sigma\rangle}$$

$$[\mathbf{GC}_2] \quad \frac{\langle gc_0, \sigma\rangle \to_1 \langle c, \sigma'\rangle}{\langle gc_0 \,[]\, gc_1, \sigma\rangle \to_1 \langle c, \sigma'\rangle} \qquad [\mathbf{GC}_3] \quad \frac{\langle gc_1, \sigma\rangle \to_1 \langle c, \sigma'\rangle}{\langle gc_0 \,[]\, gc_1, \sigma\rangle \to_1 \langle c, \sigma'\rangle}$$

$$[\mathbf{GC}_4] \quad \frac{\langle b, \sigma\rangle \to_1 \mathbf{false}}{\langle b \to c, \sigma\rangle \to_1 \mathbf{fail}} \qquad [\mathbf{GC}_5] \quad \frac{\langle gc_0, \sigma\rangle \to_1 \mathbf{fail} \quad \langle gc_1, \sigma\rangle \to_1 \mathbf{fail}}{\langle gc_0 \,[]\, gc_1, \sigma\rangle \to_1 \mathbf{fail}}$$

Note the symmetry in the rules for deriving $gc_0 \,[]\, gc_1$. The symmetry between the rules $\mathbf{GC}_2$ and $\mathbf{GC}_3$ shows that there is no preference in choosing one guarded command over the other (gc_0 or gc_1 in the command $gc_0 \,[]\, gc_1$). Hence, $gc_0 \,[]\, gc_1$ captures "pure" nondeterminism. In rule $\mathbf{GC}_5$ we see that both guarded commands gc_0 and gc_1 need to fail in order for their alternate command $gc_0 \,[]\, gc_1$ to fail (if one of them can be executed, it will). In general, all the guarded commands $gc_0, gc_1, \ldots, gc_n$ need to fail in order for $gc_0 \,[]\, gc_1 \,[]\, \ldots \,[]\, gc_n$ to fail. The semantics of **if** $(b_0 \to c_0) \,[]\, \ldots \,[]\, (b_n \to c_n)$ **fi** therefore implies that if $\neg(b_0 \vee \ldots \vee b_n)$ holds then the execution aborts.

Communicating processes

Several languages, such as CSP, Occam and Ada are available for modelling distributed systems. They all extend Hoare and Milner's pioneer work. Hoare and Milner sought for a communication primitive which will be abstract enough to model any medium used to communicate. They both independently suggested atomic actions of synchronization, with the possible exchange of values, as the central primitive of communication.

Here, we will assume process communicate via *channels*. The number of channels is known in advance; there is no creation or destruction of channels. Communication is synchronous and it is done "peer-to-peer": a process may be ready to communicate (read or write to a particular channel), however the communication will not take place until another process in its environment is ready to communicate with the complementary action (write or read from the same channel, respectively). If so, the processes will issue their read and write commands simultaneously. For example, if one process is ready to write the value of an expression a to channel α (denoted $\alpha!a$), and another process is ready to read a value from channel α and store it in location X (denoted $\alpha?X$), then after the execution of the synchronous communication, X will hold the value of a. The channels are assumed to have capacity one. That is, there is no automatic buffering: an input or output command to a certain channel is delayed until another process is ready with an output or input command to the same channel. In order to allow channels to be made local to a certain set of processes, the language supports a hiding or *restriction* command $c \setminus \alpha$ which makes α local to the communication taking place in c by hiding it from the outer world.

We now present the abstract syntax of a language of communicating processes. In addition to the sets **Loc**, **Aexp** and **Bexp** of locations, arithmetic and boolean expressions, we assume:

Channel names	$\alpha, \beta, \gamma, \ldots \in$ **Chan**
Input expressions	$\alpha?X$ where $X \in$ **Loc**
Output expression	$\alpha!a$ where $a \in$ **Aexp**

Commands

$$c ::= \textbf{skip} \mid \textbf{abort} \mid X := a \mid \alpha?X \mid \alpha!a \mid c_0; c_1 \mid \textbf{if } gc \textbf{ fi} \mid \textbf{do } gc \textbf{ od} \mid c_0 \parallel c_1 \mid c \setminus \alpha$$

Guarded commands

$$gc ::= b \to c \mid b \wedge \alpha?X \to c \mid b \wedge \alpha!a \to c \mid c_0 \,[]\, c_1$$

For simplicity we assume that the locations of each process are disjoint and the assignment command is atomic. Otherwise we would have to disallow a parallel composition $c_0 \parallel c_1$ in case c_0 and c_1 contain a common location (and any command or guarded command in which such a parallel composition occurs).

Upon executing a basic input or output command, a process needs to wait until another process is ready to perform the complementary output or input command, before it is able to proceed to execute the next command. It is thus desirable to have a mechanism that will allow probing of the channel to see if another process is ready to perform the complementary output or input command. This is the reason for having the new guarded commands $b \wedge \alpha?X \to c$ and $b \wedge \alpha!a \to c$[2].

Example.
The following example implements a process which repeatedly receives a value from the channel α and transmit it to the channel β.

$$\textbf{do } (\textbf{true} \wedge \alpha?X \to \beta!X) \textbf{ od})$$

Example.
A buffer with capacity 2 receiving on channel α and transmitting on channel γ can be implemented by the following program

$$\left(\begin{array}{l} \textbf{do } (\textbf{true} \wedge \alpha?X \to \beta!X) \textbf{ od}) \,[] \\ \textbf{do } (\textbf{true} \wedge \beta?Y \to \gamma!Y) \textbf{ od}) \end{array} \right) \setminus \beta$$

The restriction $(\ldots) \setminus \beta$ makes the channel β hidden from the environment, and thus the only communication that can be carried out in β are the ones specified above, and β may be thought of as an internal channel.

Example.
The alternative construction together with the special guarded command allow a process to "listen" simultaneously to two or more channels it expects to get an input from and read from one of them should a process in the environment wish to write there. When a read can be preformed from more than one channel, a nondeterministic choice is made between them.

[2]This construct is disallowed in Occam from practical reason.

$$
\begin{array}{l}
\textbf{if} \\
\quad (\textbf{true} \wedge \alpha?X \to c_0) \ [] \\
\quad (\textbf{true} \wedge \beta?Y \to c_1) \\
\textbf{fi}
\end{array}
$$

Compare this to the following program:

$$
\begin{array}{l}
\textbf{if} \\
\quad (\textbf{true} \to \alpha?X; c_0) \ [] \\
\quad (\textbf{true} \to \beta?Y; c_1) \\
\textbf{fi}
\end{array}
$$

The latter program may deadlock (i.e., reach a state of improper termination) while the former cannot (assuming c_0 and c_1 cannot deadlock). The latter program autonomously chooses between being prepared to receive at α or β channel. If for example it elects to right hand side (and is therefore prepared to receive at channel β), and its environment is only able to output at the α channel – there is a deadlock. Deadlocks may arise in more involved and subtle situations, such as in Dijkstra's famous examples of the "dining philosophers". For more examples see Dijkstra's book [1].

We would like to turn to the semantics for our language of communicating processes. To formally capture a synchronous communication in a rule we need a way to detect what happens in the environment. For example, if a process wishes to receive an input at channel α and store it in X, it may do so only if there is another process in its environment who wishes to write to channel α. This should be reflected in the corresponding rule. In order to save the natural locality of the processes in a distributed system, we add labels to the transitions (in a way familiar form automata theory). We define the set of labels as follows:

$$\{\alpha?n \mid \alpha \in \textbf{Chan} \ \& \ n \in \textbf{N}\} \cup \{\alpha!n \mid \alpha \in \textbf{Chan} \ \& \ n \in \textbf{N}\}$$

And we expect that the semantics will yield the following transition for example.

$$\langle \alpha?X; c_0, \sigma \rangle \xrightarrow{\alpha?n} \langle c_0, \sigma[n/X] \rangle$$

This expresses that the command $\alpha?X; c_0$ is prepared to receive a value at channel α, store it in location X (the state σ is modified accordingly) and proceed to execute command c_0. Similarly when a process is ready to output at channel α a value n held in expression e, we would expect the following transition:

$$\langle \alpha!e; c_1, \sigma \rangle \xrightarrow{\alpha!n} \langle c_1, \sigma \rangle$$

in which the state does not change. We then expect a simple handshake to have the following form:

$$\langle (\alpha?X; c_0) \parallel (\alpha!e; c_1), \sigma \rangle \longrightarrow \langle c_0 \parallel c_1, \sigma[n/X] \rangle$$

which expresses that the synchronized communication was executed. The value n of e is now held at location X and the processes are ready to execute their next commands c_0 and c_1. The transition is not labeled in this case because the communication capability of the two processes has been used up through the internal communication action. We should however expect other transitions which allow the communication to be made with an external process. This transitions can have two forms:

$$\langle (\alpha?X; c_0) \parallel (\alpha!e; c_1), \sigma \rangle \xrightarrow{\alpha?n} \langle c_0 \parallel (\alpha!e; c_1), \sigma[n/X] \rangle$$

or

$$\langle (\alpha?X; c_0) \parallel (\alpha!e; c_1), \sigma \rangle \xrightarrow{\alpha!n} \langle (\alpha?X; c_0) \parallel c_1, \sigma \rangle$$

The first transition corresponds to the situation where the left hand side received the value n from some process in the environment (rather than from the right hand side command). And the second one corresponds to the situation where the right hand side sent the value n to some process in the environment (rather than to the left hand side command).

We adopt some convention in order to simplify the presentation of the formal semantics. In order to uniformly treat both labeled and unlabeled transitions we will write $\xrightarrow{\lambda}$ with the understanding that λ ranges over labels like $\alpha?n$ and $\alpha!n$ as well as the empty label. In order to treat configurations σ and $\langle c, \sigma \rangle$ in the same way we shall regard the configuration σ as $\langle *, \sigma \rangle$ where $*$ is thought of as the *empty command* and obeys the following laws:

$$*; c \equiv c; * \equiv c \parallel * \equiv c$$

$$*; * \equiv X \parallel * \equiv (* \setminus \alpha) \equiv *$$

Last we will use $\rightarrow$ instead of $\rightarrow_1$ though we mean a small step semantics.

The formal semantics itself will be coverd in the next lecture...

Bibliography

[1] E. W. Dijkstra. *A Discipline of Programming.* Prentice Hall Series in Automatic Computation. Prentice Hall, 1976.