Establishing safety of X10 programs (v 0.1)

Vijay Saraswat

The goal of semantics is to specify a mathematically precise meaning for programs in a given programming language. In this section we will be concerned about specifying the semantics for a small imperative programming language containing the core concurrency constructs of X10.

There are two main approaches to the semantics of programming languages – the *denotational* approach, and the *operational* approach.

In the denotational approach, one identifies a mathematical space, the space of *meanings* of programs. The semantics of the programming language is then defined by a function, the *semantic function* $\mathcal{D}[\![\_]\!]$ which maps a program to an element of the mathematical space.

Consider the simple programming language defined in Table 1 . We have a set of *variables* `Var`, `x`, `y`, `z`, etc., and *values* `Val`, the integers. An *expression* is either a variable, or an arithmetic or boolean expression. A *statement* is an *assignment* `x=e` of an expression `e` to a variable `x`, or a squential composition `S1; S2`.

What should the mathematical space be. First, we must consider how to model the heap. The heap assigns a value to every valuable, so it is natural to see it as a member of the space `Heap` of functions from `Var` to `Val`. Now how do we model an expression? We must be given a heap in order to determine the value of the variables in the expression. The evaluation of the expression should return a `Val`. Hence we can think of the denotation of an expression to be a function from `Heap` to `Val`. The definition is now clear: $\mathcal{E}[\![x]\!](h) = h(x)$, and $\mathcal{E}[\![e_1 + e_2]\!] = \mathcal{E}[\![e_1]\!]\overline{+}\mathcal{E}[\![e_2]\!]$, where $\overline{+}$ represents the addition operation on integers.

Now, what should the denotation of a statement be? It starts in a heap, and after some steps, results in another heap. So it is natural then to think of modeling a statement as a *function* from `Heap` to `Heap`. For a heap h, a variable x and a value v, let $h[x = v]$ stand for the heap which is the same as h except that it takes on the value v at x. Then, clearly, the denotation of the assignment statement `x=e` should be just

$$\mathcal{S}[\![x = e]\!](h) = h[x = \mathcal{E}[\![e]\!](h)]$$

That is, evaluate e in h to obtain the value $v = \mathcal{E}[\![e]\!](h)$, and then the result is the heap $h[x = v]$.

(Variables)	x,y,z	::=	... variables ...	
(Values)	v	::=	0 | 1 | ... numbers ...	
(Expressions)	e	::=	v | x | e + e | e * e | e == e ...	
(Statements)	S,T	::=	x=e	(Assignment)
			S_0;S_1	(Sequencing)

Table 1: A simple imperative programming language

Now it should be clear what the denotation of sequential composition `S1; S2` is. It corresponds to first executing `S1` in the input heap and then computing `S2` in the resulting heap. That is:

$$\mathcal{S}[\![S_1; S_2]\!](h) = \mathcal{S}[\![S_2]\!](\mathcal{S}[\![S_1]\!](h))$$

Th denotational approach sketched above has the crucial property that the meaning of a compound phrase (e.g. `S1; S2` is given in terms of the meaning of its component phrases (e.g. `S1` and `S2`). The approach is said to be *compositional* in nature.

1 Structural operational semantics

The goal of operational semantics is to directly model the actual step-wise execution of the program.

1.1 Basic approach

First we identify a set of *configurations*. A configuration is an abstract representation of machine state. A configuration should reflect both the control and the data aspect of the computation.

Second we identify a binary transition relation $\longrightarrow$ on configurations. If $\gamma \longrightarrow \gamma'$ we think that the machine in state γ can take a single step and move to state γ'.

An *execution sequence* is a sequence $\gamma_0, \gamma_1, \gamma_2, \ldots$ such that for each i, $\gamma_i \longrightarrow \gamma_{i+1}$. The *root* of such a sequence is γ_0. We also say that γ_0 *has* the execution sequence $\gamma_0, \gamma_1, \gamma_2, \ldots$

We say that a configuration γ *diverges* if it has an infinite execution sequence. A divergent sequence represents a non-terminating execution.

We say that a configuration γ is *stuck* if there is no configuration γ' such that $\gamma \longrightarrow \gamma'$. Sometimes we will write $\gamma \not\longrightarrow$ to indicate that. A *maximal execution sequence* is a finite execution sequence whose last configuration is stuck.

A stuck configuration may represent a terminated computation or an error (e.g. a deadlocked computation). We identify a subset of stuck configurations as *terminal*. These represent the properly terminated computations. The states that are stuck but not terminal represent "bad" states – typically states the programmer did not intend for his/her program to get into. A (finite) execution sequence is *terminal* if its last configuration is terminal.

Definition 1.1 (Transition System) *A* transition system *is a triple*

$$\langle \Gamma, T, \rightarrow \rangle$$

such that Γ *is a set (of* configurations*),* $T \subseteq \Gamma$ *is the subset of* terminal *configurations and* $\longrightarrow \subseteq \Gamma \times \Gamma$ *is a binary relation on* Γ *satisfying the condition that for every* $\gamma \in T$ *there is no* γ' *such that* $\gamma \longrightarrow \gamma'$*.*

A *terminating execution sequence* from γ is an execution sequence $\gamma = \gamma_0, \gamma_1, \ldots, \gamma_n$ such that for all $\gamma_i \longrightarrow \gamma_{i+1}$ for all $i < n$, and γ_n is terminal.

The *result* of a configuration γ_0 is the set of all terminal configurations γ such that γ_0 has an execution sequence terminating in γ.

1.2 Semantics of expression evaluation

First we identify a set of *values*. For simplicity we shall take `Val`, the set of values, to be `Int`, the set of all integers. We also assume a pre-defined set of *variables*, `Var`. By a *heap* σ we mean a function from `Var` to `Val`.

Next we identify a set of *expressions*. An expression is either a value, a variable or a sum or product or an equality. (We shall assume equality returns 0 if the condition is true and 1 if it is false.) Other primitive operations can be dealt with in the same fashion.

We choose the space of configurations to be pairs $\langle e, \sigma \rangle$ or singletons v. The first represents an expression e that is intended to be evaluated in a heap σ and the second represents the result of the execution. We take the set of terminal configuration to be the singletons v.

We now provide a simple evaluator for expressions. This evaluator evaluates expressions from left to right and yields a value. Below we use "op" to stand for any of the binary operations $+$, $*$ or $==$.

$$\frac{\sigma(x) = v}{\langle x, \sigma \rangle \longrightarrow v} \qquad \frac{\langle e_0, \sigma \rangle \longrightarrow \langle e_0', \sigma' \rangle \mid v}{\langle e_0 \ op \ e_1, \sigma \rangle \longrightarrow \langle e_0' \ op \ e_1, \sigma' \rangle \mid \langle v \ op \ e_1, \sigma' \rangle}$$

$$\frac{\langle e, \sigma \rangle \longrightarrow \langle e', \sigma' \rangle \mid w \ (u = v \ op \ w)}{\langle v \ op \ e, \sigma' \rangle \longrightarrow \langle v \ op \ e', \sigma' \rangle \mid u}$$

(Variables)	x,y,z	::=	... variables ...	
(Values)	v	::=	0 | 1 | ... numbers ...	
(Expressions)	e	::=	v | x | e + e | e * e | e == e ...	
(Statements)	S,T	::=	x=e	(Assignment)
			`if` (c){S_0}`else`{S_1}	(Conditional)
			S_0;S_1	(Sequencing)
			`atomic` S	(Atomic)
			`when`(e){S}	(When)
			`async` S	(Async)
			`finish` S	(Finish)

Table 2: L0 with concurrency constructs

The rules encode a left-to-right evaluation strategy because it is not possible to evaluate the right subexpression of an expression unless the left subexpression has already been evaluated to a value.

Exercise 1.1 (No deadlock, no divergence) *Establish that given any expression e and heap σ such that all the variables in e are defined in σ, all maximal transition sequences starting from s end in a terminal configuration. Thus, there are no stuck configurations. (*Hint: use structural induction.*)*

Exercise 1.2 (Determinacy) *Establish that given a configuration $\gamma = \langle e, \sigma \rangle$ if $\gamma \longrightarrow \gamma'$ and $\gamma \longrightarrow \gamma'$ then $\gamma = \gamma'$. (*Hint: The rules encode a left-to-right evaluation strategy.*)*

That is, for any expression and heap, there is a unique transition sequence evaluating that expression into a value.

Semantics We associate with a statement S and an initial heap σ the heap σ' such that $\langle S, \sigma \rangle \longrightarrow^* \sigma'$. From the above propositions, there are no stuck configurations, and given a $\langle S, \sigma \rangle$, the terminal configuration σ' defined as above is unique.

1.3 Semantics of statements

Statements are built up from assignments, sequential composition, and conditionals using the familiar concurrency primitives (Table 3).

The transition relation for statements is given in Table 3.

$$\frac{e \longrightarrow_\sigma v}{\langle x = e, \sigma\rangle \longrightarrow \sigma[x \mapsto v]}$$

$$\frac{\langle S_0, \sigma\rangle \longrightarrow \langle S_0', \sigma'\rangle \mid \sigma'}{\langle S_0; S_1, \sigma\rangle \longrightarrow \langle S_0'; S_1, \sigma'\rangle \mid \langle S_1, \sigma'\rangle}$$

$$\frac{\langle S, \sigma\rangle \longrightarrow \langle S', \sigma'\rangle \mid \sigma'}{\langle \texttt{async}\ S, \sigma\rangle \longrightarrow \langle \texttt{async}\ S', \sigma'\rangle \mid \sigma'}$$

$$\frac{\langle S_1, \sigma\rangle \longrightarrow \langle S_1', \sigma'\rangle \mid \sigma'}{\langle \texttt{async}\ S_0; S_1, \sigma\rangle \longrightarrow \langle \texttt{async}\ S_0; S_1', \sigma'\rangle \mid \langle \texttt{async}\ S_0, \sigma'\rangle}$$

$$\frac{\langle S, \sigma\rangle \longrightarrow \langle S', \sigma'\rangle \mid \sigma'}{\langle \texttt{finish}\ S, \sigma\rangle \longrightarrow \langle \texttt{finish}\ S', \sigma'\rangle \mid \sigma'}$$

$$\frac{\langle S, \sigma\rangle \longrightarrow^* \sigma'}{\langle \texttt{atomic}\ S, \sigma\rangle \longrightarrow \sigma'}$$

$$\frac{c \longrightarrow_\sigma^* 0 \quad \langle S, \sigma\rangle \longrightarrow^* \sigma'}{\langle \texttt{when}(c)\{S\}, \sigma\rangle \longrightarrow \sigma'}$$

$$\frac{\langle e, \sigma\rangle \longrightarrow \langle e', \sigma'\rangle \mid 0 \mid 1}{\langle \texttt{if}(e)\{S_0\}\texttt{else}\{S_1\}, \sigma\rangle \longrightarrow \langle \texttt{if}(e')\{S_0\}\texttt{else}\{S_1\}, \sigma'\rangle \mid \langle S_0, \sigma'\rangle \mid \langle S_1, \sigma'\rangle}$$

Table 3: Rules defining transition relation for statements

Exercise 1.3 *Show that there are statements which have multiple terminating exection sequences with different terminal configurations.*

This shows that statements may have indeterminate execution.

Exercise 1.4 *Show that there are maximal execution sequences in which the final configuration is not terminal.*

This shows that statement execution may deadlock.

Exercise 1.5 *Show that if a statement has no subexpression of the form* `when`$(c)\{S\}$ *then all its maximal execution sequences are terminating.*

Thus, `when`$(c)\{S\}$ is the only construct that can cause a deadlock.

1.4 Statically sequential programs

A program is sequential if it does not contain `atomic`, `when`, `async` or `finish` constructs.

Exercise 1.6 *Establish that sequential statements have single maximal execution sequences which terminate in a terminal configuration.*