

# Establishing safety of X10 programs (v 0.3)

Vijay Saraswat

## 1 Denotational Semantics

### 1.1 Preliminaries

We shall be concerned with *multisets*. A multiset is a set which permits multiple occurrences of the same element. We can think of a multiset as a set where each element has an associated “hidden” tag which serves to distinguish it from all other elements. We will write a multiset with elements  $a, b, \dots, z$  as  $\{a, b, \dots, z\}$ . Thus we will not have  $\{1, 1\} = \{1\}$  since the cardinality of the left hand side is two and of the right hand side one. As usual, if  $A$  and  $B$  are two multisets we shall say  $A$  is a subset of  $B$  (and write  $A \subseteq B$  if every element of  $A$  is also an element of  $B$ ).

Let  $S$  be some multiset. An *order* over  $S$  is a transitive binary relation on  $S$ . An order is typically written using the symbol  $<$  as an infix operator. We shall be interested in *irreflexive orders*: these have the property that  $s \not< s$ , for all  $s \in S$ . Such an order  $<$  is *total* if for any two distinct elements  $a, b \in S$  either  $a < b$  or  $b < a$ ; otherwise it is *partial*.

We shall assume given a fixed set of *variables*,  $\text{Var}$  and of *values*,  $\text{Val}$ . A *heap*,  $h$ , is a function from  $\text{Var}$  to  $\text{Val}$ .

**Definition 1.1 (Step)** *A step is a pair of functions ( $c : \text{Heap} \rightarrow \text{Boolean}$ ,  $f : \text{Heap} \rightarrow \text{Heap}$ ),  $c$  is called the condition and  $f$  the action. We shall let TRUE stand for the condition that is always true (regardless of the input heap).*

*We shall let STEP stand for the space of all steps (over the fixed sets Var and Val).*

For a step  $z = (a, b)$  we let  $c_z$  stand for  $a$  and  $f_z$  for  $b$ . We will use “ $\lambda$ ” notation to write functions:

( $\lambda$ -term)  $M ::=$

$x$  a variable

$\lambda x.M$  a function with body  $M$  and formal  $x$

$(MN)$  a function application

The term  $\lambda x.M$  represents a function which when applied to value  $v$  returns the result of evaluating  $M$  with  $x$  replaced by  $v$ . The term  $(MN)$  represents the application of the function  $M$  to the value  $N$ .

**Example 1.1** Consider the statement  $x = 3$ . We shall associate with it the step  $(\text{TRUE}, f)$  where  $f$  takes as input a heap  $g$  and produces the heap which is the same as  $g$  except that the variable  $x$  is mapped to 3. We shall write such a heap as  $g[x \mapsto 3]$ , hence the action associated with  $x = 3$  is  $\lambda g.g[x \mapsto 3]$ .

A sequential program executes a totally ordered multiset of steps.<sup>1</sup> Without **async**, after each statement (step) there is a unique next statement to be executed (as recorded by the program counter). Let us say that two steps  $x$  and  $z$  are ordered by  $<$  (and write  $x < z$ ) if in every possible execution of the program  $x$  must execute before  $z$ . This order is called the *happens before* order.

In a program with **async**, at any point there may be multiple actions that could be executed ... as many as the number of **asyncs** running. Since these actions can be executed in any order, they are unordered with respect to each other. Hence the set of actions executed by a program with **asyncs** may only be partially ordered.

**Definition 1.2 (May Happen in Parallel)** For two steps  $a, b$  if neither  $a < b$  nor  $b < a$  then we say that  $a$  and  $b$  May Happen in Parallel (MHP), and write  $a \# b$ .

## 1.2 Processes

**Definition 1.3 (Process.)** A process is a triple  $P = (X, <, Z)$  where  $X$  is a (finite) multiset of steps,  $<$  is a partial order on  $X$  (the HB order) and  $Z \subseteq X$  marks the subset of synchronous steps of  $X$ , i.e. steps that are known to terminate when  $P$  (synchronously) terminates.

A process  $P = (X, <, Z)$  is said to be sequential if  $<$  is total, synchronous if  $Z = X$ , and wait-free if for every step  $x \in X$ ,  $c_x = \text{TRUE}$ .

In the literature  $(P, <)$  is called a *pomset* – a partially ordered multiset. Vaughan Pratt was one of the first researchers to emphasize the use of pomsets to model concurrency.

If  $P = (A, \ll, B)$  then we define  $X_P$  to be  $A$ ,  $<_P$  to be  $\ll$  and  $Z_P$  to be  $B$ .

For two processes  $P$  and  $Q$  we say  $P = Q$  if  $X_P = X_Q$ ,  $<_P = <_Q$  and  $Z_P = Z_Q$ .

---

<sup>1</sup>Why multiset rather than a set? Because two different statements may denote the same action.

We can define operators on processes to mimic sequential execution, `async` and `finish`. We will define the *semantic function*  $\mathcal{P}[\dots]$  which takes a statement as argument and returns a process. The use of  $[\dots]$  is conventional in denotational semantics – the brackets typically enclose syntactic elements (such as statements).

For sets  $A$  and  $B$ , the set  $A \times B$  is just the set of pairs whose first element is from  $A$  and second element from  $B$ . i.e.

$$A \times B = \{(a, b) \mid a \in A, b \in B\}$$

**Skip.** The unique process `SKIP` represents the process that does nothing. It has no steps, hence is forced to have the empty HB relation and no synchronous steps.

$$\text{SKIP} = (\emptyset, \emptyset, \emptyset) \tag{1}$$

Note it is vacuously sequential and synchronous. We shall let  $\mathcal{P}[\text{skip};] = \text{SKIP}$ .

**Basic statement.** Let  $z$  be an step. We can obtain a process from  $z$  that has the single step  $z$  as follows:

$$\text{INJECT}(z) = (\{z\}, \emptyset, \{z\}) \tag{2}$$

$\text{INJECT}(z)$  is always sequential and synchronous.

**Sequential composition.** The definition of sequential composition  $P Q$  of processes is straightforward. The steps of  $P Q$  must be precisely those of  $P$  and  $Q$ .

The HB order must respect the HB orders of  $P$  and  $Q$ , and further ensure that every step of  $Q$  is after every step of  $Z_P$ . Why  $Z_P$  instead of  $P$ ? Not all steps of  $P$  would have terminated by the time  $Q$  needs to be started – e.g.  $P$  may “contain” an `async` that needs to keep running in parallel with  $Q$ . This is precisely why we had to distinguish a subset  $Z$  of  $P$  to mark the steps of  $P$  that must terminate before subsequent processes are activated. Formally we can specify the partial order as:

$$\langle' = (\langle_P \cup \langle_Q \cup \{(p, q) \mid \exists z \in Z_P. p \langle_P z, q \in X_Q\})$$

That is,  $p \langle' q$  if and only if  $p$  HB  $q$  according to  $P$  or  $Q$  or there is some  $z$  in  $Z_P$  such that  $p$  HB  $z$  and  $z$  HB  $q$ .

The synchronous steps of  $P Q$  must be precisely those of  $P$  and  $Q$ .

So now we can define:

$$P Q = (X_P \cup X_Q, <', Z_P \cup Z_Q) \quad (3)$$

**Exercise 1.1** Show that if  $P$  and  $Q$  are processes then so is  $P Q$ .

Show that if  $P$  and  $Q$  are sequential and synchronous then so is  $P Q$ .

Show that if  $P$  and  $Q$  are sequential, but not both synchronous then  $P Q$  is neither sequential nor synchronous.

Show that  $P Q$  is wait-free precisely if  $P$  and  $Q$  are.

Note we use space (juxtaposition) in the syntax to separate two statements, and also in the semantics to specify sequential composition of processes.

**async.**  $\text{ASync}(P)$  is just like  $P$  except that it has no synchronous steps:

$$\text{ASync}(P) = (X_P, <_P, \emptyset) \quad (4)$$

$\text{ASync}(P)$  is not synchronous. It is sequential if  $P$  is. It is wait-free if  $P$  is.

**finish.**  $\text{FINISH}(P)$  is just like  $P$  except that *every* step is synchronous:

$$\text{FINISH}(P) = (X_P, <_P, X_P) \quad (5)$$

$\text{FINISH}(P)$  is synchronous. It is sequential if  $P$  is. It is wait-free if  $P$  is.

**Atomic block.** Let  $S$  be a sequential, synchronous, wait-free process. Let  $g$  represent the step obtained by composing the steps of  $X_S$  in the order specified by  $<_X$ . Let  $z = (\text{TRUE}, g)$ .

$$\text{ATOMIC}(S) = (\{z\}, \emptyset, \{z\}) \quad (6)$$

$\text{ATOMIC}(S)$  is sequential, synchronous and wait-free.

**Conditional atomic block.** Let  $S$  be a sequential, synchronous, wait-free process and let  $c : \text{Heap} \rightarrow \text{Boolean}$  be a condition. Let  $g$  represent the step obtained by composing the steps of the steps of  $X_S$  in the order specified by  $<_X$ . Let  $z = (c, g)$ .

$$\text{WHEN}(c, S) = (\{z\}, \emptyset, \{z\}) \quad (7)$$

$\text{WHEN}(c, S)$  is sequential, synchronous and wait-free only if  $c = \text{TRUE}$ .

**Exercise 1.2** Show that  $\text{ATOMIC}(S) = \text{WHEN}(\text{TRUE}, S)$ .

**Semantic function.** We can relate syntax (statements) to semantics (processes) through the semantic function  $\mathcal{P}[\dots]$ .

**Definition 1.4** We define the semantic function  $\mathcal{P}[\dots]$  that takes a statement and returns its associated process inductively as follows. It assumes a sister semantic function  $\mathcal{S}[\dots]$  that takes a basic statement  $b$  (e.g. a read or write statement or a variable declaration) and returns the action corresponding to it, or a single condition  $c$  (e.g. an equality comparison) and returns the function corresponding to it.

$$\begin{aligned}
 \mathcal{P}[\text{skip};] &= \text{SKIP} \\
 \mathcal{P}[b] &= \text{INJECT}(\mathcal{S}[b]) \\
 \mathcal{P}[s \ t] &= \mathcal{P}[s] \ \mathcal{P}[t] \\
 \mathcal{P}[\text{async } s] &= \text{ASYNC}(\mathcal{P}[s]) \\
 \mathcal{P}[\text{finish } s] &= \text{FINISH}(\mathcal{P}[s]) \\
 \mathcal{P}[\text{atomic } s] &= \text{ATOMIC}(\mathcal{P}[s]) \\
 \mathcal{P}[\text{when}(c)\{s\}] &= \text{WHEN}(\mathcal{S}[c], \mathcal{P}[s])
 \end{aligned}$$

**Example 1.2** Consider the statement  $s$  given by:

```

var x:Long=0;
finish {
  async x=2;
  async x=2;
}

```

The process associated with this  $\mathcal{P}[s] = (\{a, b, c\}, \{a < b, a < c\}, \{a, b, c\})$  where:

- $a = (\text{TRUE}, \lambda g.g[x \mapsto 0])$
- $b = (\text{TRUE}, \lambda g.g[x \mapsto 2])$
- $c = (\text{TRUE}, \lambda g.g[x \mapsto 2])$

This process is synchronous, but not sequential.

### 1.3 Properties of combinators

**Exercise 1.3** Check the following are true:

1. SKIP is a fixed point of FINISH and ASYNC:

$$\text{FINISH}(\text{SKIP}) = \text{ASYNC}(\text{SKIP}) = \text{SKIP}$$

2. SKIP is the unit of sequential composition:

$$\text{SKIP } P = P \text{ SKIP} = P$$

3. Sequential composition is associative:

$$(P \ Q) \ R = P \ (Q \ R)$$

4. ASYNC is idempotent:

$$\text{ASYNC}(\text{ASYNC}(P)) = \text{ASYNC}(P)$$

5. ASYNC distributes over sequential composition, if the first argument is asynchronous:

$$\text{ASYNC}(\text{ASYNC}(P) \ Q) = \text{ASYNC}(P) \ \text{ASYNC}(Q)$$

6. FINISH absorbs a nested FINISH or ASYNC:

$$\text{FINISH}(\text{FINISH}(P)) = \text{FINISH}(\text{ASYNC}(P)) = \text{FINISH}(P)$$

7. FINISH absorbs a FINISH or ASYNC nested in the second argument of a sequential composition:

$$\text{FINISH}(P \ \text{ASYNC}(Q)) = \text{FINISH}(P \ \text{FINISH}(Q)) = \text{FINISH}(P \ Q)$$

8. Nested atomics flatten out:

$$\text{ATOMIC}(\text{ATOMIC}(S)) = \text{ATOMIC}(S)$$

9. Atomics nested within a sequential composition flatten out:

$$\text{ATOMIC}(P \ \text{ATOMIC}(S)) = \text{ATOMIC}(\text{ATOMIC}(P) \ S) = \text{ATOMIC}(P \ S)$$

**Exercise 1.4** Give examples that illustrate the following are false:

- $\text{ASYNC}(P) \ Q = P \ \text{ASYNC}(Q)$
- $\text{FINISH}(\text{ASYNC}(P)) = P$
- $\text{FINISH}(\text{ASYNC}(P) \ \text{ASYNC}(Q)) = \text{FINISH}(P \ Q)$

## 1.4 Execution.

**Definition 1.5** An execution of a process  $P = (X, <, Z)$  is obtained by running the steps of  $X$  in any total order that extends  $<$  (from an initial heap), provided that the conditions of each step are satisfied by the heap at that step.

Formally, let  $E \subseteq X$  and  $<'$  be a total order on  $E$  extending  $<$ . Let  $z_1, \dots, z_n$  be the elements of  $E$  enumerated according to  $<'$ . Let  $h_0$  be an initial heap and for  $i > 0$ , define  $h_i = f_{z_i}(h_{i-1})$ . Then  $(E, <')$  is an execution of  $P$  if:

1.  $E$  correctly answers all conditions: for all  $i > 0$   $c_{z_i}(h_{i-1})$  is true.
2.  $E$  is maximal, the steps not in  $E$  cannot be scheduled: for all steps  $z \in X \setminus E$ ,  $c_z(h_n)$  is false.

The observation generated by an execution  $(E, <')$  is  $(h_0, h_n)$  if  $E = X$ ; otherwise it is  $(h_0, \perp)$  (indicating that this execution deadlocks). The set of observations of  $P$  is denoted by  $o(P)$ .

**Definition 1.6 (Determinate Processes)** A process  $P$  is said to be determinate if  $o(P)$  represents the graph of a function, that is, if  $o(P)$  contains two pairs  $(g, h)$  and  $(g, h')$  then it is the case that  $h = h'$ .

In other words, a determinate process will produce a unique output when run on a given heap.

**Exercise 1.5** Show: programs with `async` may have an HB order that is not total, hence may have multiple distinct executions, and therefore, multiple distinct results.

**Exercise 1.6 (Wait-free processes cannot deadlock.)** Show that if  $P$  is a wait-free process then for no heap  $h$  is it the case that  $(h, \perp) \in o(P)$ .

## 1.5 Data race.

A program  $s$  is said to have a *concrete data race* if the associated process  $\mathcal{P}[s]$  has

- two steps  $a, b$ , such that  $a \# b$
- there is a mutable location  $l$  that one of them writes and the other reads/writes.

**Theorem 1.1** *If  $s$  has no concrete data races then  $\mathcal{P}[[s]]$  is determinate.*

Lack of concrete data races is a sufficient, not a necessary condition for determinacy. Programs with races may still be determinate (cf Example 1.2).

## References