Establishing safety of X10 programs (v 0.2)

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 $a, b, \ldots, z$ as $\{a, b, \ldots, z\}$. 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*, `Var` and of *values*, `Val`. A *heap*, h, is a function from `Var` to `Val`.

Definition 1.1 (Action) *An* action *is a function that takes as input a heap g and produces as output another heap g. We shall let* Act *stand for the space of all actions (over the fixed sets* `Var` *and* `Val`*).*

We will use "λ" notation to writen functions:

(λ-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 action f which 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 actions.[1] Without `async`, after each statement (action) there is a unique next statement to be executed (as recorded by the program counter). Let us say that two actions f and g are ordered by $<$ (and write $f < g$) if in every possible execution of the program f must execute before g. 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 `async`s 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 actions 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 actions, $<$ is a partial order on X (the HB order) and $Z \subseteq X$ marks the subset of* synchronous *actions of X, i.e. actions that are known to terminate when P (synchronously) terminates.*

A process $P = (X, <, Z)$ is said to be sequential *if $<$ is total and* synchronous *if $Z = X$.*

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 $P = (X, <, Z)$ then we define X_P to be X, $<_P$ to be $<$ and Z_P to be Z.

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

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

[1] Why multiset rather than a set? Because two different statements may denote the same action.

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 actions, hence is forced to have the empty HB relation and no synchronous actions.

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

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

Basic actions. Let f be an action. We can obtain a process from f that has the single action f as follows:

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

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

Sequential composition. The definition of sequential composition $P\ Q$ of processes is straightforward. The actions 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. (Note that not all actions 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 actions of P that must terminate before subsequent processes are activated.)

The synchronous actions of $P\ Q$ must be precisely those of P and Q. So now we can define:

$$P\ Q = (X_P \cup X_Q, (<_P \cup <_Q \cup \{(p, q) \mid \exists z \in Z_P . p <_P z, q \in X_Q\}, Z_P \cup Z_Q) \tag{3}$$

Exercise 1.1 *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.

Note we use space (juxtaposition) in the syntax to separate two statements, and also in the semantics to specify sequential composition of processes. We will use brackets "{" ... "}" around processes to indicate order of application of operators (such as sequencing).

`async`. ASYNC(P) is just like P except that it has no synchronous actions:

$$\text{ASYNC}(P) = (X_P, <_P, \emptyset) \tag{4}$$

ASYNC(P) is not synchronous. It is sequential if P is.

`finish`. FINISH(P) is just like P except that *every* action is synchronous:

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

FINISH(P) is synchronous. It is sequential if P is.

Atomic block. Let S be a sequential and synchronous process. Let g represent the action obtained by composing the actions of X_S in the order specified by $<_X$.

$$\text{ATOMIC}(S) = (\{g\}, \emptyset, \{g\}) \tag{6}$$

ATOMIC(S) is sequential and synchronous.

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

Definition 1.4 *We define the semantic function $\mathcal{P}[\![\ldots]\!]$ that takes a statement and returns its associated process inductively as follows. It assumes a sister semantic function $\mathcal{S}[\![\ldots]\!]$ that takes a single statement s (e.g. a read or write statement or a variable declaration) and returns the action corresponding to it.*

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

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:*

- $a = \lambda g.g[x \mapsto 0]$
- $b = \lambda g.g[x \mapsto 2]$
- $c = \lambda g.g[x \mapsto 2]$

This process is synchronous, but not sequential.

1.3 Properties of combinators

Exercise 1.2 *Check the following are true:*

- SKIP *is a fixed point of* FINISH *and* ASYNC*:*

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

- SKIP *is the unit of sequential composition:*

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

- *Sequential composition is associative:*

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

- ASYNC *is idempotent:*

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

- ASYNC *distributes over sequential composition, if the first argument is asynchronous:*

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

- FINISH *absorbs a nested* FINISH *or* ASYNC*:*

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

- FINISH *absorbs a* FINISH *or* ASYNC *nested in the second argument of a sequential composition:*

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

Exercise 1.3 *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 actions of X in any total order that extends $<$, from an initial heap, g. The* result *of the execution is the final heap, h. We say that (g, h) is an* observation *of P. 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 1.4 *Show: programs with* `async` *may have an HB order that is not total, hence may have multiple distinct executions, and therefore, multiple distinct results.*

1.5 Data race.

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

- two actions 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).