Operational Semantics of DistAlgo

Scott D. Stoller
August 18, 2014

This document gives an abstract syntax and operational semantics for a core language for DistAlgo [LSLG12]. The operational semantics is a reduction semantics with evaluation contexts [WF94, ŠRM09].

1 Abstract Syntax

The abstract syntax is defined in Figures 1 and 2. We use some syntactic sugar in sample code. Specifically, (1) we use infix notation for some binary operators, such as and is; (2) in comprehensions, if the Boolean expression is true, we elide “| true”; (3) in await statements, if the statement in one of the clauses is skip, we elide “: skip”.

Notation used in the grammar.

• A symbol in the grammar is a terminal symbol if it starts with a lower-case letter.
• A symbol in the grammar is a non-terminal symbol if it starts with an upper-case letter.
• | separates alternatives.
• * after a non-terminal means “0 or more occurrences”
• + after a non-terminal means “1 or more occurrences”

Well-formedness requirements on programs.

1. The top-level method in a program must be named main. It gets executed in an instance of the Process class when the program starts.

2. Each label used in a receive definition must be the label of some statement that appears in the same class as the receive definition.

3. Invocations of methods defined using def appear only in method call statements. Invocations of methods defined using defun appear only in method call expressions.

Constructs whose semantics is given by translation.

1. The semantics of await statements not preceded by explicitly specified labels is given by translation. Specifically, for each await-statement whose associated label is the empty string, we generate a fresh label name ℓ, associate ℓ with that await statement (by replacing the empty string with ℓ in that await statement), and insert ℓ in every at-clause in the class containing the await-statement.
Program ::= Configuration ProcessClass* Method

ProcessClass ::= class ClassName extends ClassName: Method* ReceiveDef*

ReceiveDef ::= receive ReceivePattern+ at LabelName+ : Statement
| receive ReceivePattern+ : Statement

ReceivePattern ::= Pattern from InstanceVariable

Method ::= def MethodName(Parameter*) Statement
| defun MethodName(Parameter*) Expr

Statement ::= Label InstanceVariable = Expr
| Label InstanceVariable = new ClassName
| Label InstanceVariable = { Pattern : Iterator* | Expr }
| Label Statement ; Statement
| Label if Expr: Statement else: Statement
| Label for Iterator: Statement
| Label while Expr: Statement
| Label Expr.MethodName(Expr*)
| Label send Tuple to Expr
| Label await Expr : Stmt AnotherAwaitClause*
| Label await Expr : Stmt AnotherAwaitClause* timeout Expr
| Label skip

Expr ::= Literal
| Parameter
| InstanceVariable
| Tuple
| Expr.MethodName(Expr*)
| UnaryOp(Expr)
| BinaryOp(Expr,Expr)
| isinstance(Exp,ClassName)
| and(Expr,Expr) // conjunction (short-circuiting)
| or(Expr,Expr) // disjunction (short-circuiting)
| each Iterator | Expr
| some Iterator | Expr

Tuple ::= (Expr*)

Figure 1: Abstract Syntax, Part 1

2. The boolean operators and and each can be statically eliminated as follows, so we assume in the semantics below that the program does not contain these constructs: $e_1$ and $e_2$ is replaced with not(not($e_1$) or not($e_2$)), and each $iter$ | $e$ is replaced with not(some $iter$ | not($e$)).
UnaryOp ::=  
| not  // boolean negation
| isTuple  // test whether a value is a tuple
| len  // length of a tuple

BinaryOp ::=  
| is  // identity-based equality, as in Python
| plus  // sum
| select  // select(t,i) returns the i’th component of tuple t

Pattern ::= InstanceVariable | TuplePattern

TuplePattern ::= (PatternElement*)

PatternElement ::= Constant | InstanceVariable | =InstanceVariable

Iterator ::= Pattern in Expr

AnotherAwaitClause ::= or Expr : Stmt

Configuration ::= configuration ChannelOrder ChannelReliability ...
ChannelOrder ::= fifo | unordered
ChannelReliability ::= reliable | unreliable

ClassName ::= ...
MethodName ::= ...
Parameter ::= self | ...
InstanceVariable ::= Expression.Field
Field ::= ...
Label ::= emptyString | LabelName
LabelName ::= ...
Literal ::= BooleanLiteral | IntegerLiteral | ...
BooleanLiteral ::= true | false
IntegerLiteral ::= ...

Figure 2: Abstract Syntax, Part 2

3. Comprehensions can be statically eliminated as follows, so we assume in the semantics below that the program does not contain comprehensions. Specifically, the comprehension \( \ell x = \{ e \mid x_1 \text{ in } e_1, \ldots, x_n \text{ in } e_n \mid b \} \), where \( \ell \) is a label and each \( x_i \) is a pattern, can be replaced with

\[
\ell x = \text{new Set()}
\text{for } x_1 \text{ in } e_1:
\ldots
\text{for } x_n \text{ in } e_n:
\text{if } b:
\quad x\text{.add}(e)
\]

4. Iterators containing tuple patterns can be statically rewritten as iterators without tuple patterns, so
we assume in the semantics below that iterators do not contain tuple patterns. The rewriting is done as follows.

- Consider the universal quantification each \((e_1, \ldots, e_n)\) in \(s \mid b\). Let \(\theta\) be the substitution that replaces \(e_i\) with \(\text{select}(x, i)\) for each \(i\) such that \(e_i\) is a variable not prefixed with “\(=\)”. Let \(\{j_1, \ldots, j_m\}\) contain the indices of the constants and the variables prefixed with “\(=\)” in \((e_1, \ldots, e_n)\). Let \(x\) be a fresh variable. The quantification can be rewritten as each \(x\) in \(s \mid \text{isTuple}(x)\) and \(\text{len}(x) = n\) and \(\text{select}(x,j_1) = e_{j_1}\) and \(\ldots\) and \(\text{select}(x,j_m) = e_{j_m}\) implies \(b\theta\).

 Informally, the semantics is that values in \(s\) that do not match the tuple pattern are ignored by the universal quantification, i.e., \(b\) does not need to hold for those values.

- Consider the existential quantification some \((e_1, \ldots, e_n)\) in \(s \mid b\). Let \(\theta\) be the substitution that replaces \(e_i\) with \(\text{select}(x, i)\) for each \(i\) such that \(e_i\) is a variable not prefixed with “\(=\)”. Let \(\{j_1, \ldots, j_m\}\) contain the indices of the constants and the variables prefixed with “\(=\)” in \((e_1, \ldots, e_n)\). Let \(x\) be a fresh variable. The quantification can be rewritten as some \(x\) in \(s \mid \text{isTuple}(x)\) and \(\text{len}(x) = n\) and \(\text{select}(x,j_1) = e_{j_1}\) and \(\ldots\) and \(\text{select}(x,j_m) = e_{j_m}\) and \(b\theta\).

- Consider the loop \(\text{for } (e_1, \ldots, e_n) \in e : s\). Let \(\{i_1,\ldots,i_k\}\) contain the indices in \((e_1, \ldots, e_n)\) of variables not prefixed with “\(=\)”. Let \(\{j_1, \ldots, j_m\}\) contain the indices in \((e_1, \ldots, e_n)\) of constants and variables prefixed with “\(=\)”. Let \(\theta\) be the substitution that replaces \(e_i\) with \(\text{select}(x, i)\) for each \(i\) in \(\{i_1, \ldots, i_k\}\). Let \(x\) and \(y\) be fresh variables. Note that \(e\) may denote a set or sequence, and duplicate bindings for the tuple of variables \((e_{i_1}, \ldots, e_{i_k})\) are filtered out if \(e\) is a set but not if \(e\) is a sequence. The loop can be rewritten as

\[
\text{if } \text{isinstance}(x,\text{Set}) \\\quad // \text{filter out duplicate bindings. } y \text{ contains processed bindings.} \\
y = \text{new } \text{Set} \\
\text{for } x \text{ in } e: \\
\quad \text{if } (\text{isTuple}(x) \text{ and len}(x) = n) \\
\quad \quad \text{and } \text{select}(x,j_1) = e_{j_1} \text{ and } \ldots \\
\quad \quad \text{and } \text{select}(x,j_m) = e_{j_m} \text{ and } b\theta \\
\quad \quad \text{and not } y.\text{contains}((\text{select}(x,i_1), \ldots, \text{select}(x,i_k))): \\
\quad \quad \quad y.\text{add}((\text{select}(x,i_1), \ldots, \text{select}(x,i_k))) \\
\quad \quad s\theta \\
\quad \quad \text{else:} \\
\quad \quad \quad \text{skip} \\
\quad \quad \text{else} \\
\quad \quad \quad \text{for } x \text{ in } e: \\
\quad \quad \quad \text{if } (\text{isTuple}(x) \text{ and len}(x) = n) \\
\quad \quad \quad \quad \text{and } \text{select}(x,j_1) = e_{j_1} \text{ and } \ldots \\
\quad \quad \quad \quad \text{and } \text{select}(x,j_m) = e_{j_m} \text{ and } b\theta): \\
\quad \quad \quad \quad s\theta \\
\quad \quad \quad \text{else:} \\
\quad \quad \quad \quad \text{skip} \\
\text{else:} \\
\quad \text{skip}
\]

Notes.

1. **ClassName** must include **Process**. **Process** is a pre-defined class. It should not be defined explicitly. **Process** has fields **sent** and **received**, and it has a method **start**.
2. The grammar allows receive definitions to appear in classes that do not extend \texttt{Process}, but such receive definitions are useless, so it would be reasonable to declare them illegal.

3. Sets and sequences are treated as objects, because they are mutable. \texttt{ClassName} must include \texttt{Set} and \texttt{Sequence}. These are predefined classes that should not be defined explicitly. Methods of \texttt{Set} include \texttt{add}, \texttt{del}, \texttt{contains}, \texttt{min}, \texttt{max}, and \texttt{size}. Methods of \texttt{Sequence} include \texttt{add} (which adds an element at the end of the sequence), \texttt{contains}, and \texttt{length}. We give the semantics explicitly for a few of these methods; the others are handled similarly.

4. Tuples are treated as values, not as objects, because they are immutable.

5. Object allocation and comprehension are statements, not expressions, because they have side-effects. All expressions are side-effect free.

6. The values of method parameters cannot be updated (e.g., using assignment statements). For brevity, local variables of methods are omitted from the language. Consequently, assignment is allowed only for instance variables.

7. Semantically, the \texttt{for} loop copies the contents of a (mutable) sequence or set into an (immutable) tuple before iterating over it, to ensure that changes to the sequence or set by the loop body do not affect the iteration. An implementation could use optimizations to achieve this semantics without copying when possible.

8. For brevity, among the standard arithmetic operations (+, -, *, etc.), we include only one representative operation in the abstract syntax and semantics; others are handled similarly.

9. The semantics below does not model real-time, so timeouts in \texttt{await} statements are simply allowed to occur non-deterministically.

10. We omit the concept of “site” (process location) from the semantics, and we omit the site argument from the constructor when creating instances of process classes, because process location does not affect other aspects of the semantics.

11. We do not include \texttt{use handling_all} explicitly in the program syntax, but we assume its effect in the semantics, since it is the default.

12. To support initialization of a process by its parent, a process can access fields of another process and invoke methods on another process before the latter process is started but not afterward. In the future, we might extend the semantics to allow remote method invocation, i.e., to allow a process to invoke methods on another process after the latter process is started.

13. We require that all messages are tuples. This is an inessential restriction; it slightly simplifies the specification of pattern matching between messages and receive patterns.

14. A process’s \texttt{sent} sequence contains pairs of the form \((m, d)\), where \(m\) is a message sent by the process to destination \(d\). A process’s \texttt{receive} sequence contains pairs of the form \((m, s)\), where \(m\) is a message received by the process from sender \(s\).

## 2 Semantic Domains

The semantic domains are defined in Figure 3.
\begin{align*}
\text{Bool} &= \{\text{true}, \text{false}\} \\
\text{Int} &= \ldots \\
\text{Address} &= \ldots \\
\text{ProcessAddress} &= \ldots \\
\text{Tuple} &= \text{Val}^* \\
\text{Val} &= \text{Bool} \cup \text{Int} \cup \text{Address} \cup \text{Tuple} \\
\text{Object} &= (\text{Field} \rightarrow \text{Val}) \cup \text{SetOfVal} \cup \text{SeqOfVal} \\
\text{SetOfVal} &= \text{Set}(\text{Val}) \\
\text{SeqOfVal} &= \text{Val}^* \\
\text{MsgQueue} &= (\text{Tuple} \times \text{ProcessAddress})^* \\
\text{ChannelStates} &= \text{ProcessAddress} \times \text{ProcessAddress} \\
&\quad \rightarrow \text{Tuple}^* \\
\text{HeapType} &= \text{Address} \rightarrow \text{Name} \\
\text{LocalHeap} &= \text{Address} \rightarrow \text{Object} \\
\text{Heap} &= \text{ProcessAddress} \rightarrow \text{LocalHeap} \\
\text{State} &= (\text{ProcessAddress} \rightarrow \text{Statement}) \\
&\quad \times \text{HeapType} \times \text{Heap} \times \text{ChannelStates} \\
&\quad \times (\text{ProcessAddress} \rightarrow \text{MsgQueue})
\end{align*}

Figure 3: Semantic Domains

**Notation:**

- $D^*$ contains sequences of values from domain $D$.
- $\text{Set}(D)$ contains sets of values from domain $D$.
- $D_1 \rightarrow D_2$ contains partial functions from $D_1$ to $D_2$. $\text{dom}(f)$ is the domain of a partial function $f$.

**Notes.**

- We require $\text{ProcessAddress} \subseteq \text{Address}$.
- For $a \in \text{ProcessAddress}$ and $h \in \text{Heap}$, $h(a)$ is the local heap of process $a$. For $a \in \text{Address}$ and $ht \in \text{HeapType}$, $ht(a)$ is the type of the object with address $a$. For convenience, we use a single (global) function for $\text{HeapType}$ in the semantics, even though the information in that function is distributed in the same way as the heap itself in an implementation.
- The $\text{MsgQueue}$ associated with a process by the last component of a state contains messages (paired with their senders) that have arrived at the process but have not yet been received. This information is needed to express the requirement that all matching messages that have arrived at the process must be handled when execution of the process reaches a yield point.
Expression ::= 
  | Address 
  | Address.Field 

Statement ::= 
  | for Variable intuple Tuple: Statement 

Figure 4: Extensions to the abstract syntax 

C ::= 
  [] 
  | [Val*,C,Expr*] 
  | Expr.MethodName(Val*,C,Expr*) 
  | UnaryOp(C) 
  | BinaryOp(C,Expr) 
  | BinaryOp(Val,C) 
  | isinstance(C,ClassName) 
  | or(C,Expr) 
  | some Pattern in C | Expr 
  | C.Field = Expression 
  | Address.Field = C 
  | InstanceVariable = C 
  | C ; Statement 
  | if C: Statement else: Statement 
  | for InstanceVariable in C: Statement 
  | for InstanceVariable intuple Tuple: C 
  | send C to Expr 
  | send Val to C 
  | await Expr : Stmt AnotherAwaitClause* timeout C 

Figure 5: Evaluation contexts 

3  Extended Abstract Syntax 

Section 1 defines the abstract syntax of programs that can be written by the user. Figure 4 extends the abstract syntax to include additional forms into which programs may evolve during evaluation. Only the new productions are shown here; all of the productions given above carry over unchanged.

The statement for \( v \) intuple \( t \): \( s \) iterates over the elements of tuple \( t \), in the obvious way.

4  Evaluation Contexts 

Evaluation contexts, also called reduction contexts, are used to identify the next part of an expression or statement to be evaluated. An evaluation context is an expression or statement with a hole, denoted [], in place of the next sub-expression or sub-statement to be evaluated. Evaluation contexts are defined in Figure 5.
5 Transition Relations

The transition relation for expressions has the form $e \rightarrow_{ht,h} e'$, where $e$ and $e'$ are expressions, $ht \in \text{HeapType}$, and $h \in \text{LocalHeap}$.

The transition relation for statements has the form $\sigma \rightarrow \sigma'$ where $\sigma \in \text{State}$ and $\sigma' \in \text{State}$. Both transition relations are implicitly parameterized by the program, which is needed to look up method definitions and configuration information. The transition relation for expressions is defined in Figure 6. The transition relation for statements is defined in Figures 7–9.

Notation and auxiliary functions.

- In the transition rules, $a$ matches an address; $p$ matches a primitive value (i.e., a boolean value, integer value, or an address); $v$ matches a value (i.e., a primitive value or a tuple whose components are values); and $\ell$ matches a label.

- For an expression or statement $e$, $e[x := y]$ denotes $e$ with all occurrences of $x$ replaced with $y$.

- A function matches the pattern $f[x \rightarrow y]$ if $f(x)$ equals $y$.

- $f[x := y]$ denotes the function that is the same as $f$ except that it maps $x$ to $y$.

- $f_0$ denotes the empty partial function, i.e., the partial function whose domain is the empty set.

- For a (partial) function $f$, $f \odot a$ denotes the function that is the same as $f$ except that it has no mapping for $a$.

- Sequences are denoted with angle brackets, e.g., $(0, 1, 2) \in \text{Int}^*$.

- $s@t$ is the concatenation of sequences $s$ and $t$.

- $\text{tail}(s)$ is the tail of sequence $s$, i.e., the sequence obtained by removing the first element of $s$.

- $\text{first}(s)$ is the first element of sequence $s$.

- $\text{length}(s)$ is the length of sequence $s$.

- $\text{extends}(c_1, c_2)$ holds iff class $c_1$ is a descendant of class $c_2$ in the inheritance hierarchy.

- For $c \in \text{ClassName}$,
  $$\text{new}(c) = \begin{cases} \{\} & \text{if } c = \text{Set} \\ \langle \rangle & \text{if } c = \text{Sequence} \\ f_0 & \text{otherwise} \end{cases}$$

- For $m \in \text{MethodName}$ and $c \in \text{ClassName}$, $\text{methodDef}(c, m, \text{def})$ holds if (1) class $c$ defines method $m$, and $\text{def}$ is the definition of $m$ in $c$, or (2) $c$ does not define $m$, and $\text{def}$ is the definition of $m$ in the nearest ancestor of $c$ in the inheritance hierarchy that defines $m$.

- For $h, \bar{h}, \bar{h}' \in \text{LocalHeap}$ and $ht, ht' \in \text{HeapType}$ and $v, v' \in \text{Val}$, $\text{isCopy}(v, h, \bar{h}, ht, v', \bar{h}', ht')$ holds if (1) $v$ is a value for a process with local heap $h$ (i.e., addresses in $v$ are evaluated with respect to $h$), (2) $v'$ is a copy of $v$ for a process with local heap $\bar{h}'$, i.e., $v'$ is the same as $v$ except that, instead of referencing objects in $h$, it references newly allocated copies of those objects in $\bar{h}'$, and (3) $\bar{h}'$ and $ht'$ are versions of $\bar{h}'$ and $ht$ updated to reflect the allocation of those objects. As an exception, because process addresses are used as global identifiers, process addresses in $v$ are copied unchanged into $v'$. 

8
and new copies of process objects are not created. We give some auxiliary definitions and then a formal
definition of isCopy. For $v \in Val$, let $addrs(v, h)$ denote the set of addresses that appear in $v$ or in any
objects or values reachable from $v$ with respect to local heap $h$; formally,

$$a \in addrs(v, h) = v \in Address \land v = a$$
$$\forall v \in dom(h) \land h(v) \in Field \rightarrow Val \land (\exists f \in dom(h(v)).a \in addrs(h(v)(f), h))$$
$$\forall v \in dom(h) \land h(v) \in (SetOfVal \cup SeqOfVal) \land (\exists v' \in h(v).a \in addrs(v', h))$$
$$\forall (v_1, \ldots, v_n) \in Val. \exists i \in [1..n]. v = (v_1, \ldots, v_n) \land a \in addrs(v_i, h))$$

For $v, v' \in Val$ and $f \in Address \rightarrow Address$, $subst(v, v', f)$ holds if $v$ is obtained from $v'$ by replacing
each occurrence of an address $a$ in $dom(f)$ with $f(a)$ (informally, $f$ maps addresses of new objects in
$v'$ to addresses of corresponding old objects in $v$); formally,

$$subst(v, v', f) = v \in Bool \cup Int \cup (Address \setminus dom(f)) \land v' = v$$
$$\forall v \in dom(f) \land f(v') = v$$
$$\forall (\exists v_1, \ldots, v_n, v'_1, \ldots, v'_n, v = (v_1, \ldots, v_n) \land \forall i \in [1..n]. subst(v_i, v'_i, f))$$

Similarly, for $o, o' \in Object$ and $f \in Address \rightarrow Address$, $subst(o, o', f)$ holds if $o$ is obtained from $o'$ by replacing
each occurrence of an address $a$ in $dom(f)$ with $f(a)$. For sets $S$ and $S'$, let $S \sqsupseteq S'$ be the set of bijections between $S$ and $S'$. Finally, isCopy is defined as follows (informally, $A$ contains the
addresses of the newly allocated objects):

$$isCopy(v, h, h, ht, v', h', ht') = \exists A \subset Address \setminus ProcessAddress, f \in A^{1-1} \ (addrs(v, h) \setminus ProcessAddress) \land$$
$$\forall h \in dom(ht) = A \land dom(ht') = dom(h) \cup A \land dom(h') = dom(h) \cup A \land$$
$$\forall a \in dom(ht). ht'(a) = ht(a) \land$$
$$\forall a \in A. ht'(a) = ht(f(a)) \land subst(h(a), h'(a), f))$$

- For $m \in Val, a \in ProcessAddress, \ell \in LabelName, h \in LocalHeap$, and a receive definition $d$, if message
$m$ can be received from $a$ at label $\ell$ by a process with local heap $h$ using receive definition $d$, then
$matchRcvDef(m, a, \ell, h, d)$ returns the appropriately instantiated body of $d$. Specifically, if (1) either
$d$ lacks an at clause, or $d$ has an at clause that includes $\ell$ (note: this implies that $\ell$ is a label name
and is not the empty string), and (2) $d$ contains a receive pattern $P$ from $x$ such that there exists a
substitution $\theta$ such that (2a) $m = P\theta$ and (2b) $\theta(y) = h(y)$ for every variable $y$ prefixed with “$=$” in
$P$, then, letting $\theta$ be the substitution obtained using the first receive pattern in $d$ for which (2) holds,
$matchRcvDef(m, a, \ell, h, d)$ returns $s\theta[x := a]$, where $s$ is the body of $d$ (i.e., the statement that appears
in $d$). Otherwise, $matchRcvDef(m, a, \ell, d)$ returns $\bot$.

- For $m, \tilde{m} \in Val, a \in ProcessAddress, \ell \in LabelName, c \in ClassName$, and $h \in LocalHeap$, if message
$m$ can be received from $a$ at label $\ell$ in class $c$ by a process with local heap $h$, and $\tilde{m}$ is a copy of $m$, then
$rcvAtLabel(m, \tilde{m}, a, \ell, c, h)$ returns a statement that should be executed when receiving $m$ in that
context. Specifically, if class $c$ contains a receive definition $d$ such that
$matchRcvDef(m, a, \ell, h, d)$ is not $\bot$, then, letting $d_1, \ldots, d_n$ be the sequence of receive definitions $d$ in $c$ (in the order that they appear in $c$) such that
$matchRcvDef(m, a, \ell, h, d)$ is not $\bot$, and letting $s_i = matchRcvDef(m, \tilde{m}, a, \ell, h, d_i)$,
$rcvAtLabel(m, \tilde{m}, a, \ell, c, h)$ returns $self.received.add((\tilde{m}, a))$: $s_1; \ldots; s_n$. Otherwise, if $\ell$ is not the
empty string, and there does not exist a receive definition $d$ in $c$ and a label $\ell'$ such that
$matchRcvDef(m, a, \ell', h, d)$ holds, then $rcvAtLabel(m, \tilde{m}, a, \ell, c, h)$ returns $self.received.add((\tilde{m}, a))$. Otherwise, $rcvAtLabel(m, \tilde{m}, a, \ell, c, h)$ returns $\bot$. 

9
For $q \in \text{MsgQueue}$, $\ell \in \text{LabelName}$, $c \in \text{ClassName}$, and $h \in \text{LocalHeap}$, if $q$ contains an entry $(m, \bar{m}, a)$ such that $\text{rcvAtLabel}(m, \bar{m}, a, \ell, c, h)$ is not $\perp$, then, letting $(m, \bar{m}, a)$ be the first such entry in $q$, $\text{rcvMsg}(q, \ell, c, h)$ returns $(q', s)$, where $q'$ is obtained from $q$ by removing $(m, \bar{m}, a)$, and $s$ is the statement returned by $\text{rcvAtLabel}(m, \bar{m}, a, \ell, c, h)$. Otherwise, $\text{rcvMsg}(q, \ell, c, h)$ returns $\perp$.

6 Executions

An execution is a sequence of transitions $\sigma_0 \rightarrow \sigma_1 \rightarrow \sigma_2 \rightarrow \cdots$ such that $\sigma_0$ is an initial state. The set of initial states is defined in Figure 10. Intuitively, $a_p$ is the address of the initial process, $a_r$ is the address of the received sequence of the initial process, and $a_s$ is the address of the sent sequence of the initial process.

Informally, execution of the statement initially associated with a process may eventually (1) terminate (i.e., the statement associated with the process becomes $\text{skip}$, indicating that there is nothing left for the process to do), (2) get stuck (i.e., the statement associated with the process is not $\text{skip}$, and the process has no enabled transitions) due to an unsatisfied $\text{await}$ statement or an error (e.g., the statement contains an expression that tries to select a component from a value that is not a tuple) or (3) run forever due to an infinite loop or infinite recursion.

Acknowledgments

The semantics is based on discussions with Yanhong Liu, DistAlgo’s main designer. Bo Lin and Xuetian Weng provided helpful suggestions.

References


% primitive value
\( p \rightarrow_{ht,h} p \)

% field access
\( a.f \rightarrow_{ht,h} h(a)(f) \) if \( a \in \text{dom}(h) \land f \in \text{dom}(h(a)) \)

% tuple
\((v_1, \ldots, v_n) \rightarrow_{ht,h} (v_1, \ldots, v_n)\)

% invoke method in user-defined class
\( a.m(v_1, \ldots, v_n) \rightarrow_{ht,h} e[\text{self} := a, x_1 := v_1, \ldots, x_n := v_n] \)
if \( \exists c. \ ht(a) = c \land \text{methodDef}(c, m, \text{defun} m(x_1, \ldots, x_n) e) \)

% invoke method in pre-defined class (representative examples)
\( a.\text{contains}(v_1) \rightarrow_{ht,h} \text{true} \) if \( \exists S. \ ht(a) = \text{Set} \land h(a) = S \land v_1 \in S \)
\( a.\text{contains}(v_1) \rightarrow_{ht,h} \text{false} \) if \( \exists S. \ ht(a) = \text{Set} \land h(a) = S \land v_1 \notin S \)

% unary operations
\( \text{not}(\text{true}) \rightarrow_{ht,h} \text{false} \)
\( \text{not}(\text{false}) \rightarrow_{ht,h} \text{true} \)
\( \text{isTuple}(v) \rightarrow_{ht,h} \text{true} \) if \( v \) is a tuple
\( \text{isTuple}(v) \rightarrow_{ht,h} \text{false} \) if \( v \) is not a tuple
\( \text{len}(v) \rightarrow_{ht,h} n \) if \( v \) is a tuple with \( n \) components

% binary operations
\( \text{is}(v_1, v_2) \rightarrow_{ht,h} \text{true} \) if \( v_1 \) and \( v_2 \) are the same value
\( \text{plus}(v_1, v_2) \rightarrow_{ht,h} v_3 \) if \( v_1 \in \text{Int} \land v_2 \in \text{Int} \land v_3 = v_1 + v_2 \)
\( \text{select}(v_1, v_2) \rightarrow_{ht,h} v_3 \)
if \( v_2 \in \text{Int} \land v_2 > 0 \land (v_1 \text{ is a tuple with at least } v_2 \text{ components}) \land (v_3 \text{ is the } v_2'\text{th component of } v_1) \)

% instanceof
\( \text{instance}(a, c) \rightarrow_{ht,h} \text{true} \) if \( h(t(a)) = c \)
\( \text{instance}(a, c) \rightarrow_{ht,h} \text{false} \) if \( h(t(a)) \neq c \)

% disjunction
\( \text{or}(\text{true}, e) \rightarrow_{ht,h} \text{true} \)
\( \text{or}(\text{false}, e) \rightarrow_{ht,h} e \)

% existential quantification
\( \text{some } x \text{ in } a \mid e \rightarrow_{ht,h} e[x := v_1] \lor \cdots \lor e[x := v_n] \)
if \( (ht(a) = \text{Sequence} \land h(a) = \langle v_1, \ldots, v_n \rangle) \lor (ht(a) = \text{Set} \land \langle v_1, \ldots, v_n \rangle \text{ is a linearization of } h(a)) \)

Figure 6: Transition relation for expressions.
% field assignment
(P[a → ℓ a′.f = v], ht, h[a → ha[a′ → o]], ch, mq) → (P[a := skip], ht, h[a := ha[a′ := o[f := v]]], ch, mq)
if rcvMsg(mq(a), ℓ, ht(a), h(a)) = ⊥

% object allocation
(P[a → ℓ a′.f = new c], ht, h[a → ha[a′ → o]], ch, mq)
→ (P[a := skip], ht[a′ := c], h[a := ha[a′ := o[f := a]], a := new(c)], ch, mq)
if ac ∉ dom(ht) ∧ ac ∈ Address ∧ (ac ∈ ProcessAddress ↔ extends(c, Process))
∧ rcvMsg(mq(a), ℓ, ht(a), h(a)) = ⊥

% sequential composition
(P[a → ℓ skip•s], ht, h, ch, mq) → (P[a := s], ht, h, ch, mq)
if rcvMsg(mq(a), ℓ, ht(a), h(a)) = ⊥

% conditional statement
(P[a → ℓ if true: s₁ else : s₂], ht, h, ch, mq) → (P[a := s₁], ht, h, ch, mq)
if rcvMsg(mq(a), ℓ, ht(a), h(a)) = ⊥

(P[a → ℓ if false: s₁ else : s₂], ht, h, ch, mq) → (P[a := s₂], ht, h, ch, mq)
if rcvMsg(mq(a), ℓ, ht(a), h(a)) = ⊥

% for loop
(P[a → ℓ for x in a′: s], ht, h, ch, mq) → (P[a := for x intuple (v₁,..., vₙ): s], ht, h, ch, mq)
if (ht(a) = Sequence ∧ h(a)(a′) = (v₁,..., vₙ)) ∨ (ht(a) = Set ∧ (v₁,..., vₙ) is a linearization of h(a)(a′))
∧ rcvMsg(mq(a), ℓ, ht(a), h(a)) = ⊥

(P[a → ℓ for x intuple (v₁,..., vₙ): s], ht, h, ch, mq)
→ (P[a := s[x := v₁]: for x intuple (v₂,..., vₙ): s], ht, h, ch, mq)
(P[a → ℓ for x intuple (): s], ht, h, ch, mq) → (P[a := skip], ht, h, ch, mq)

% while loop
(P[a → ℓ while e: s], ht, h, ch, mq) → (P[a := if e: (s; while e: s) else : skip], ht, h, ch, mq)
if rcvMsg(mq(a), ℓ, ht(a), h(a)) = ⊥

% invoke method in user-defined class
(P[a → ℓ a′.m(v₁,..., vₙ)], ht, h, ch, mq) → (P[a := s[self := a, x₁ := v₁,..., xₙ := vₙ]], ht, h, ch, mq)
if a′ ∈ dom(h(a))
∧ ht(a′) ∉ {Process, Set, Sequence} ∧ methodDef(ht(a′), m, def m(x₁,..., xₙ) s)
∧ rcvMsg(mq(a), ℓ, ht(a), h(a)) = ⊥

% invoke method in pre-defined class (representative examples)

% Process.start allocates a local heap and sent and received sequences for the new process,
% and moves the started process to the new local heap
(P[a → ℓ a′.start()], ht, h[a → ha[a′ → o]], ch, mq)
→ (P[a := skip, a′ := a′.run()], ht[a := Sequence, a := Sequence],
h[a := ha ⊔ a′, a := fₙ[a′ → o], sent := a response := a, received := ()), ch, mq)
if extends(ht(a′), Process) ∧ (ht(a′) inherits start from Process) ∧ a′ ∉ dom(ht) ∧ a' ∉ dom(ht)
∧ a'' ∈ Address \ ProcessAddress ∧ a'' ∈ Address \ ProcessAddress ∧ rcvMsg(mq(a), ℓ, ht(a), h(a)) = ⊥

Figure 7: Transition relation for statements, part 1.
% invoke method in pre-defined class (representative examples, continued)
(P[a → ℓ .add(v1)], ht, h[a → ha], ch, mq) → (P[a := skip], ht, h[a := ha := ha(a') ∪ {v1}], ch, mq)
if a' ∈ dom(ha) ∧ ht(a') = Set ∧ rcvMsg(mq(a), ℓ, ht(a), h(a)) = ⊥

(P[a → ℓ .append(v1)], ht, h[a → ha], ch, mq) → (P[a := skip], ht, h[a := ha := ha(a')@ha(v1)], ch, mq)
if a' ∈ dom(ha) ∧ ht(a') = Sequence ∧ ht(v1) = Sequence ∧ rcvMsg(mq(a), ℓ, ht(a), h(a)) = ⊥

% send to a process. Create copies of the message for the sender's send history and the receiver.
(P[a → ℓ send v to a'], ht, h[a → ha], ch, mq)
→ (P[a := skip], ht2, h[a := ha1[a_s := ha(a_s)@((v1, a'))]], a' := ha1[1], ch[(a, a') := (v')@ch((a, a'))], mq)
if a' ∈ ProcessAddress ∧ a_s = ha(a)(sent) ∧ isCopy(v, ha, ht, v1, ha1, ht1)
∧ isCopy(v, ha1, ha', ht1, v', ha1, ht2) ∧ rcvMsg(mq(a), ℓ, ht(a), h(a)) = ⊥

% send to a set of processes
(P[a → ℓ send v to a'], ht, h[a → ha], ch, mq)
→ (P[a := for x in a' send v to x], ht, h[a := ha(a_s := ha(a_s)@((v, a')))], ch, mq)
if ht(a') = Set ∧ a_s = ha(a)(sent) ∧ (x is a fresh variable) ∧ rcvMsg(mq(a), ℓ, ht(a), h(a)) = ⊥

% message reordering
(P, ht, h, ch[(a, a') → q], mq) → (P, ht, h, ch[(a, a') := q'], mq)
if (channel order is unordered in the program configuration) ∧ (q' is a permutation of q)

% message loss
(P, ht, h, ch[(a, a') → q], mq) → (P, ht, h, ch[(a, a') := q'], mq)
if (channel reliability is unreliable in the program configuration) ∧ (q' is a subsequence of q)

% receive a message
(P[a → ℓ s], ht, h, ch, mq[a → q]) → (P[a := s'[self := a]; ℓ s], ht, h, ch, mq[a := q'])
if rcvMsg(mq(a), ℓ, ht(a), h(a)) = (q', s')

% arrival of message at process (message moves from channel to message queue)
% append tuple (m, m', a) to message queue. m' is a copy of message m for storage in receive history.
(P, ht, h[a' → ha], ch[(a, a') → q], mq)
→ (P, ht', h[a' → ha'], ch[(a, a') := tail(q)], mq[a' := mq(a')@((first(q), m', a))])
if length(q) > 0 ∧ isCopy(first(q), ha, ha, m, m', ha', ht')

% await without timeout clause
(P[a → ℓ await e1; s1 or · · · or en; s_n], ht, h, ch, mq) → (P[a := s_i], ht, h, ch, mq)
if i ∈ [1..n] ∧ ei →ht,h(a) true ∧ rcvMsg(mq(a), ℓ, ht(a), h(a)) = ⊥

% await with timeout clause, terminated by true condition
(P[a → ℓ await e1; s1 or · · · or en; s_n timeout v;s], ht, h, ch, mq) → (P[a := s_i], ht, h, ch, mq)
if i ∈ [1..n] ∧ ei →ht,h(a) true ∧ rcvMsg(mq(a), ℓ, ht(a), h(a)) = ⊥

% await with timeout clause, terminated by timeout (occurs non-deterministically)
(P[a → ℓ await e1; s1 or · · · or en; s_n timeout v;s], ht, h, ch, mq) → (P[a := s_i], ht, h, ch, mq)
if e1 →ht,h(a) false ∧ · · · ∧ en →ht,h(a) false ∧ rcvMsg(mq(a), ℓ, ht(a), h(a)) = ⊥

Figure 8: Transition relation for statements, part 2.
% context rule for expressions
\[ e \rightarrow_{ht,h(a)} e' \]
\[(P[a \rightarrow C[e]], h, t, h, ch, mq) \rightarrow (P[a := C[e']], h, t, h, ch, mq)\]

% context rule for statements
\[(P[a \rightarrow s], h, t, h, ch, mq) \rightarrow (P[a := s'], h', t', h', ch', mq')\]
\[(P[a \rightarrow C[s]], h, t, h, ch, mq) \rightarrow (P[a := C[s']], h', t', h', ch', mq')\]

Figure 9: Transition relation for statements, part 3.

\[ \text{Init} = \{ (P, h, t, h, ch, mq) \in \text{State} \mid \]
\[ \exists a_p \in \text{ProcessAddress}, \]
\[ a_r \in \text{Address} \setminus \text{ProcessAddress}, \]
\[ a_s \in \text{Address} \setminus \text{ProcessAddress}. \]
\[ a_r \neq a_s \land \]
\[ P = f_0[a_p := \text{main}] \land \]
\[ ht = f_0[a_p := \text{Process}, a_r := \text{Sequence}, a_s := \text{Sequence}] \land \]
\[ h = f_0[a_p := h] \land \]
\[ ch = (\lambda (a_1, a_2) \in \text{ProcessAddress} \times \text{ProcessAddress}. () ) \land \]
\[ mq = (\lambda a \in \text{ProcessAddress}. () ) \land \]
\[ \text{where } ha = f_0[a_p := o_p, a_r := (), a_s := ()] \]
\[ o_p = f_0[\text{received} := a_r, \text{sent} := a_s] \}

Figure 10: Initial states.