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, ŞRM09].

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` 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 ℓ x `= {` e `|` x_1 `in` e_1`,` ..., x_n `in` e_n `|` b }, where ℓ is a label and each x_i is a pattern, can be replaced with

```
ℓ x = new Set()
for $x_1$ in $e_1$:
  ...
    for $x_n$ in $e_n$:
      if $b$:
        $x$.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 `|` b. Let θ be the substitution that replaces e_i with `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 `| (isTuple(`x`) and len(`x`) is` n `and select(`x`,`j_1`) is` e_{j_1} `and` $\cdots$ `and select(`x`,`j_m`) is` 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 `|` b. Let θ be the substitution that replaces e_i with `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 `| isTuple(`x`) and len(`x`) is` n `and select(`x`,`j_1`) is` e_{j_1} `and` $\cdots$ `and select(`x`,`j_m`) is` e_{j_m} `and` $b\theta$.
- Consider the loop `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 θ be the substitution that replaces e_i with `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

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

else
  for $x$ in $e$ :
    if (isTuple($x$) and len($x$) is $n$
        and select($x$,$j_1$) is $e_{j_1}$ and $\cdots$
        and select($x$,$j_m$) is $e_{j_m}$ and $b\theta$):
          $s\theta$
    else:
          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 `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. `ClassName` must include `Set` and `Sequence`. These are predefined classes that should not be defined explicitly. Methods of `Set` include `add`, `del`, `contains`, `min`, `max`, and `size`. Methods of `Sequence` include `add` (which adds an element at the end of the sequence), `contains`, and `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 `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 `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 `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 `sent` sequence contains pairs of the form (m, d), where m is a message sent by the process to destination d. A process's `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.

$$
\begin{array}{rcl}
\mathit{Bool} & = & \{\texttt{true}, \texttt{false}\} \\
\mathit{Int} & = & ... \\
\mathit{Address} & = & ... \\
\mathit{ProcessAddress} & = & ... \\
\mathit{Tuple} & = & \mathit{Val}^* \\
\mathit{Val} & = & \mathit{Bool} \cup \mathit{Int} \cup \mathit{Address} \cup \mathit{Tuple} \\
\mathit{Object} & = & (\texttt{Field} \rightharpoonup \mathit{Val}) \cup \mathit{SetOfVal} \cup \mathit{SeqOfVal} \\
\mathit{SetOfVal} & = & \mathrm{Set}(\mathit{Val}) \\
\mathit{SeqOfVal} & = & \mathit{Val}^* \\
\mathit{MsgQueue} & = & (\mathit{Tuple} \times \mathit{ProcessAddress})^* \\
\mathit{ChannelStates} & = & \mathit{ProcessAddress} \times \mathit{ProcessAddress} \\
 & & \rightharpoonup \mathit{Tuple}^* \\
\mathit{HeapType} & = & \mathit{Address} \rightharpoonup \texttt{ClassName} \\
\mathit{LocalHeap} & = & \mathit{Address} \rightharpoonup \mathit{Object} \\
\mathit{Heap} & = & \mathit{ProcessAddress} \rightharpoonup \mathit{LocalHeap} \\
\mathit{State} & = & (\mathit{ProcessAddress} \rightharpoonup \texttt{Statement}) \\
 & & \times \mathit{HeapType} \times \mathit{Heap} \times \mathit{ChannelStates} \\
 & & \times (\mathit{ProcessAddress} \rightharpoonup \mathit{MsgQueue})
\end{array}
$$

Figure 3: Semantic Domains

Notation:

- D^* contains sequences of values from domain D.
- $\mathrm{Set}(D)$ contains sets of values from domain D.
- $D1 \rightharpoonup D2$ contains partial functions from D_1 to D_2. $dom(f)$ is the domain of a partial function f.

Notes.

- We require $\mathit{ProcessAddress} \subseteq \mathit{Address}$.
- For $a \in \mathit{ProcessAddress}$ and $h \in \mathit{Heap}$, $h(a)$ is the local heap of process a. For $a \in \mathit{Address}$ and $ht \in \mathit{HeapType}$, $ht(a)$ is the type of the object with address a. For convenience, we use a single (global) function for *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 *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, $ht \in \mathit{HeapType}$, and $h \in \mathit{LocalHeap}$.

The transition relation for statements has the form $\sigma \rightarrow \sigma'$ where $\sigma \in \mathit{State}$ and $\sigma' \in \mathit{State}$. 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 ℓ 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, $f \ominus a$ 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., $\langle 0, 1, 2\rangle \in \mathit{Int}^*$.
- $s@t$ is the concatenation of sequences s and t.
- $\mathit{tail}(s)$ is the tail of sequence s, i.e., the sequence obtained by removing the first element of s.
- $\mathit{first}(s)$ is the first element of sequence s.
- $\mathit{length}(s)$ is the length of sequence s.
- $\mathit{extends}(c_1, c_2)$ holds iff class c_1 is a descendant of class c_2 in the inheritance hierarchy.
- For $c \in \texttt{ClassName}$,
$$\mathit{new}(c) = \begin{cases} \{\} & \text{if } c = \texttt{Set} \\ \langle\rangle & \text{if } c = \texttt{Sequence} \\ f_0 & \text{otherwise} \end{cases}$$
- For $m \in \texttt{MethodName}$ and $c \in \texttt{ClassName}$, $\mathit{methodDef}(c, m, \mathit{def})$ holds if (1) class c defines method m, and def is the definition of m in c, or (2) c does not define m, and def is the definition of m in the nearest ancestor of c in the inheritance hierarchy that defines m.
- For $h, \bar{h}, \bar{h}' \in \mathit{LocalHeap}$ and $ht, ht' \in \mathit{HeapType}$ and $v, v' \in \mathit{Val}$, $\mathit{isCopy}(v, h, \bar{h}, ht, v', \bar{h}', ht')$ 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',

and new copies of process objects are not created. We give some auxiliary definitions and then a formal definition of *isCopy*. For $v \in \mathit{Val}$, let $\mathit{addrs}(v,h)$ 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,

$$\begin{aligned} a \in \mathit{addrs}(v,h) = {} & v \in \mathit{Address} \wedge v = a \\ & \vee v \in \mathit{dom}(h) \wedge h(v) \in \texttt{Field} \rightharpoonup \mathit{Val} \wedge (\exists f \in \mathit{dom}(h(v)).a \in \mathit{addrs}(h(v)(f),h)) \\ & \vee v \in \mathit{dom}(h) \wedge h(v) \in (\mathit{SetOfVal} \cup \mathit{SeqOfVal}) \wedge (\exists v' \in h(v).a \in \mathit{addrs}(v',h)) \\ & \vee (\exists v_1,\ldots,v_n \in \mathit{Val}.\ \exists i \in [1..n].\ v = (v_1,\ldots,v_n) \wedge a \in \mathit{addrs}(v_i,h)) \end{aligned}$$

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

$$\begin{aligned} \mathit{subst}(v,v',f) = {} & v \in \mathit{Bool} \cup \mathit{Int} \cup (\mathit{Address} \setminus \mathit{dom}(f) \wedge v' = v \\ & \vee v \in \mathit{dom}(f) \wedge f(v') = v \\ & \vee (\exists v_1,\ldots,v_n,v'_1,\ldots,v'_n.\ v = (v_1,\ldots,v_n) \wedge v' = (v'_1,\ldots,v'_n) \wedge (\forall i \in 1..n.\ \mathit{subst}(v_i,v'_i,f))) \end{aligned}$$

Similarly, for $o, o' \in \mathit{Object}$ and $f \in \mathit{Address} \rightharpoonup \mathit{Address}$, $\mathit{subst}(o,o',f)$ holds if o is obtained from o' by replacing each occurrence of an address a in $\mathit{dom}(f)$ with $f(a)$. For sets S and S', let $S \overset{1-1}{\to} S'$ 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):

$$\begin{aligned} \mathit{isCopy}(v,h,\bar{h},ht,v',\bar{h}',ht') = {} & \exists A \subset \mathit{Address} \setminus \mathit{ProcessAddress}, f \in A \overset{1-1}{\to} (\mathit{addrs}(v,h) \setminus \mathit{ProcessAddress}). \\ & A \cap \mathit{dom}(ht) = \emptyset \wedge \mathit{dom}(ht') = \mathit{dom}(ht) \cup A \wedge \mathit{dom}(\bar{h}') = \mathit{dom}(\bar{h}) \cup A \\ & \wedge (\forall a \in \mathit{dom}(ht).\ ht'(a) = ht(a)) \\ & \wedge (\forall a \in \mathit{dom}(\bar{h}).\ \bar{h}'(a) = \bar{h}(a)) \\ & \wedge (\forall a \in A.\ ht'(a) = ht(f(a)) \wedge \mathit{subst}(\bar{h}(a),\bar{h}'(a),f)) \end{aligned}$$

- For $m \in \mathit{Val}$, $a \in \mathit{ProcessAddress}$, $\ell \in \texttt{LabelName}$, $h \in \mathit{LocalHeap}$, and a receive definition d, if message m can be received from a at label ℓ by a process with local heap h using receive definition d, then $\mathit{matchRcvDef}(m,a,\ell,h,d)$ returns the appropriately instantiated body of d. Specifically, if (1) either d lacks an `at` clause, or d has an `at` clause that includes ℓ (note: this implies that ℓ 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 θ such that (2a) $m = P\theta$ and (2b) $\theta(y) = h(y)$ for every variable y prefixed with "`=`" in P, then, letting θ be the substitution obtained using the first receive pattern in d for which (2) holds, $\mathit{matchRcvDef}(m,a,\ell,h,d)$ returns $s\theta[x := a]$, where s is the body of d (i.e., the statement that appears in d). Otherwise, $\mathit{matchRcvDef}(m,a,\ell,d)$ returns $\perp$.

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

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

6 Executions

An execution is a sequence of transitions $\sigma_0 \to \sigma_1 \to \sigma_2 \to \cdots$ such that σ_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 `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 `skip`, and the process has no enabled transitions) due to an unsatisfied `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

[LSLG12] Yanhong A. Liu, Scott D. Stoller, Bo Lin, and Michael Gorbovitski. From clarity to efficiency for distributed algorithms. In *Proceedings of the 2012 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 395–410. ACM Press, October 2012.

[ŞRM09] Traian Florin Şerbănuţă, Grigore Roşu, and José Meseguer. A rewriting logic approach to operational semantics. *Information and Computation*, 207:305–340, 2009.

[WF94] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115:38–94, 1994.

% primitive value
$p \rightarrow_{ht,h} p$

% field access
$a.f \rightarrow_{ht,h} h(a)(f) \quad$ if $a \in \mathit{dom}(h) \wedge f \in \mathit{dom}(h(a))$

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

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

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

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

% binary operations
$\texttt{is}(v_1, v_2) \rightarrow_{ht,h} \texttt{true} \quad$ if v_1 and v_2 are the same value

$\texttt{plus}(v_1, v_2) \rightarrow_{ht,h} v_3 \quad$ if $v_1 \in \mathit{Int} \wedge v_2 \in \mathit{Int} \wedge v_3 = v_1 + v_2$

$\texttt{select}(v_1, v_2) \rightarrow_{ht,h} v_3$
if $v_2 \in \mathit{Int} \wedge v_2 > 0 \wedge$ (v_1 is a tuple with at least v_2 components) $\wedge$ (v_3 is the v_2'th component of v_1)

% isinstance
$\texttt{isinstance}(a, c) \rightarrow_{ht,h} \texttt{true} \quad$ if $ht(a) = c$
$\texttt{isinstance}(a, c) \rightarrow_{ht,h} \texttt{false} \quad$ if $ht(a) \neq c$

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

% existential quantification
$\texttt{some}\ x\ \texttt{in}\ a \mid e \quad \rightarrow_{ht,h} \quad e[x := v_1]\ \texttt{or}\ \cdots\ \texttt{or}\ e[x := v_n]$
if $(ht(a) = \texttt{Sequence} \wedge h(a) = \langle v_1, \ldots, v_n \rangle) \vee (ht(a) = \texttt{Set} \wedge \langle v_1, \ldots, v_n \rangle$ is a linearization of $h(a))$

Figure 6: Transition relation for expressions.

% field assignment
$(P[a \to \ell\ a'.f = v], ht, h[a \to ha[a' \to o]], ch, mq) \to (P[a := \texttt{skip}], ht, h[a := ha[a' := o[f := v]]], ch, mq)$
if $\mathit{rcvMsg}(mq(a), \ell, ht(a), h(a)) = \bot$

% object allocation
$(P[a \to \ell\ a'.f = \texttt{new}\ c], ht, h[a \to ha[a' \to o]], ch, mq)$
$\to (P[a := \texttt{skip}], ht[a' := c], h[a := ha[a' := o[f := a_c], a_c := \mathit{new}(c)]], ch, mq)$
if $a_c \notin \mathit{dom}(ht) \wedge a_c \in \mathit{Address} \wedge (a_c \in \mathit{ProcessAddress} \iff \mathit{extends}(c, \texttt{Process}))$
$\wedge\ \mathit{rcvMsg}(mq(a), \ell, ht(a), h(a)) = \bot$

% sequential composition
$(P[a \to \ell\ \texttt{skip};s], ht, h, ch, mq) \to (P[a := s], ht, h, ch, mq)$
if $\mathit{rcvMsg}(mq(a), \ell, ht(a), h(a)) = \bot$

% conditional statement
$(P[a \to \ell\ \texttt{if true} : s_1\ \texttt{else} : s2], ht, h, ch, mq) \to (P[a := s_1], ht, h, ch, mq)$
if $\mathit{rcvMsg}(mq(a), \ell, ht(a), h(a)) = \bot$

$(P[a \to \ell\ \texttt{if false} : s_1\ \texttt{else} : s2], ht, h, ch, mq) \to (P[a := s_2], ht, h, ch, mq)$
if $\mathit{rcvMsg}(mq(a), \ell, ht(a), h(a)) = \bot$

% for loop
$(P[a \to \ell\ \texttt{for}\ x\ \texttt{in}\ a'\texttt{:}\ s], ht, h, ch, mq) \to (P[a := \texttt{for}\ x\ \texttt{intuple}\ (v_1, \ldots, v_n) : s], ht, h, ch, mq)$
if $((ht(a) = \texttt{Sequence} \wedge h(a)(a') = \langle v_1, \ldots, v_n\rangle) \vee (ht(a) = \texttt{Set} \wedge \langle v_1, \ldots, v_n\rangle$ is a linearization of $h(a)(a')))$
$\wedge\ \mathit{rcvMsg}(mq(a), \ell, ht(a), h(a)) = \bot$

$(P[a \to \texttt{for}\ x\ \texttt{intuple}\ (v_1, \ldots, v_n) : s], ht, h, ch, mq)$
$\to (P[a := s[x := v_1]; \texttt{for}\ x\ \texttt{intuple}\ (v_2, \ldots, v_n) : s], ht, h, ch, mq)$

$(P[a \to \texttt{for}\ x\ \texttt{intuple}\ () : s], ht, h, ch, mq) \to (P[a := \texttt{skip}], ht, h, ch, mq)$

% while loop
$(P[a \to \ell\ \texttt{while}\ e\texttt{:}\ s], ht, h, ch, mq) \to (P[a := \texttt{if}\ e\texttt{:}\ (s;\ \texttt{while}\ e\texttt{:}\ s)\ \texttt{else} : \texttt{skip}], ht, h, ch, mq)$
if $\mathit{rcvMsg}(mq(a), \ell, ht(a), h(a)) = \bot$

% invoke method in user-defined class
$(P[a \to \ell\ a'.m(v_1, \ldots, v_n)], ht, h, ch, mq) \to (P[a := s[\texttt{self} := a, x_1 := v_1, \ldots, x_n := v_n]], ht, h, ch, mq)$
if $a' \in \mathit{dom}(h(a))$
$\wedge\ ht(a') \notin \{\texttt{Process}, \texttt{Set}, \texttt{Sequence}\} \wedge \mathit{methodDef}(ht(a'), m, \texttt{def}\ m(x_1, \ldots, x_n)\ s)$
$\wedge\ \mathit{rcvMsg}(mq(a), \ell, ht(a), h(a)) = \bot$

% 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
$(P[a \to \ell\ a'.\texttt{start}()], ht, h[a \to ha[a' \to o], ch, mq)$
$\to (P[a := \texttt{skip}, a' := a'.\texttt{run}()], ht[a_s := \texttt{Sequence}, a_r := \texttt{Sequence}],$
$h[a := ha \ominus a', a' := f_0[a' \to o[\texttt{sent} := a_s, \texttt{received} := a_r], a_r := \langle\rangle, a_s := \langle\rangle]], ch, mq)$
if $\mathit{extends}(ht(a'), \texttt{Process}) \wedge (ht(a')$ inherits `start` from `Process`$) \wedge a_r \notin \mathit{dom}(ht) \wedge a_s \notin \mathit{dom}(ht)$
$\wedge\ a_r \in \mathit{Address} \setminus \mathit{ProcessAddress} \wedge a_s \in \mathit{Address} \setminus \mathit{ProcessAddress} \wedge \mathit{rcvMsg}(mq(a), \ell, ht(a), h(a)) = \bot$

Figure 7: Transition relation for statements, part 1.

% invoke method in pre-defined class (representative examples, continued)
$(P[a \to \ell\ a'.\texttt{add}(v_1)], ht, h[a \to ha], ch, mq) \to (P[a := \texttt{skip}], ht, h[a := ha[a' := ha(a') \cup \{v_1\}]], ch, mq)$
if $a' \in \mathit{dom}(ha) \wedge ht(a') = \texttt{Set} \wedge \mathit{rcvMsg}(mq(a), \ell, ht(a), h(a)) = \bot$

$(P[a \to \ell\ a'.\texttt{append}(v_1)], ht, h[a \to ha], ch, mq) \to (P[a := \texttt{skip}], ht, h[a := ha[a' := ha(a')@ha(v_1)]], ch, mq)$
if $a' \in \mathit{dom}(ha) \wedge ht(a') = \texttt{Sequence} \wedge ht(v_1) = \texttt{Sequence} \wedge \mathit{rcvMsg}(mq(a), \ell, ht(a), h(a)) = \bot$

% send to a process. create copies of the message for the sender's send history and the receiver.
$(P[a \to \ell\ \texttt{send}\ v\ \texttt{to}\ a'], ht, h[a \to ha, a' \to ha'], ch, mq)$
$\to (P[a := \texttt{skip}], ht_2, h[a := ha_1[a_s := ha(a_s)@\langle(v_1, a')\rangle], a' := ha'_1], ch[(a, a') := \langle v'\rangle @ ch((a, a'))], mq)$
if $a' \in \mathit{ProcessAddress} \wedge a_s = ha(a)(\texttt{sent}) \wedge \mathit{isCopy}(v, ha, ha, ht, v_1, ha_1, ht_1)$
$\wedge\ \mathit{isCopy}(v, ha_1, ha', ht_1, v', ha'_1, ht_2) \wedge \mathit{rcvMsg}(mq(a), \ell, ht(a), h(a)) = \bot$

% send to a set of processes
$(P[a \to \ell\ \texttt{send}\ v\ \texttt{to}\ a'], ht, h[a \to ha], ch, mq)$
$\to (P[a := \texttt{for}\ x\ \texttt{in}\ a'\texttt{:}\ \texttt{send}\ v\ \texttt{to}\ x], ht, h[a := ha[a_s := ha(a_s)@\langle(v, a')\rangle]], ch, mq)$
if $ht(a') = \texttt{Set} \wedge a_s = ha(a)(\texttt{sent}) \wedge (x \text{ is a fresh variable}) \wedge \mathit{rcvMsg}(mq(a), \ell, ht(a), h(a)) = \bot$

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

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

% receive a message
$(P[a \to \ell\ s], ht, h, ch, mq[a \to q]) \to (P[a := s'[\texttt{self} := a]; \ell\ s], ht, h, ch, mq[a := q'])$
if $\mathit{rcvMsg}(mq(a), \ell, ht(a), h(a)) = (q', s')$

% 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.
$(P, ht, h[a' \to ha], ch[(a, a') \to q], mq)$
$\to (P, ht', h[a' \to ha'], ch[(a, a') := \mathit{tail}(q)], mq[a' := mq(a')@\langle(\mathit{first}(q), m', a)\rangle)$
if $\mathit{length}(q) > 0 \wedge \mathit{isCopy}(\mathit{first}(q), ha, ha, ht, m', ha', ht')$

% await without timeout clause
$(P[a \to \ell\ \texttt{await}\ e_1{:}s_1\ \texttt{or}\ \cdots\ \texttt{or}\ e_n{:}s_n], ht, h, ch, mq) \to (P[a := s_i], ht, h, ch, mq)$
if $i \in [1..n] \wedge e_i \to_{ht,h(a)} \texttt{true} \wedge \mathit{rcvMsg}(mq(a), \ell, ht(a), h(a)) = \bot$

% await with timeout clause, terminated by true condition
$(P[a \to \ell\ \texttt{await}\ e_1{:}s_1\ \texttt{or}\ \cdots\ \texttt{or}\ e_n{:}s_n\ \texttt{timeout}\ v{:}s], ht, h, ch, mq) \to (P[a := s_i], ht, h, ch, mq)$
if $i \in [1..n] \wedge e_i \to_{ht,h(a)} \texttt{true} \wedge \mathit{rcvMsg}(mq(a), \ell, ht(a), h(a)) = \bot$

% await with timeout clause, terminated by timeout (occurs non-deterministically)
$(P[a \to \ell\ \texttt{await}\ e_1{:}s_1\ \texttt{or}\ \cdots\ \texttt{or}\ e_n{:}s_n\ \texttt{timeout}\ v{:}s], ht, h, ch, mq) \to (P[a := s], ht, h, ch, mq)$
if $e_1 \to_{ht,h(a)} \texttt{false} \wedge \cdots \wedge e_n \to_{ht,h(a)} \texttt{false} \wedge \mathit{rcvMsg}(mq(a), \ell, ht(a), h(a)) = \bot$

Figure 8: Transition relation for statements, part 2.

% context rule for expressions

$$\frac{e \rightarrow_{ht,h(a)} e'}{(P[a \rightarrow C[e]], ht, h, ch, mq) \rightarrow (P[a := C[e']], ht, h, ch, mq)}$$

% context rule for statements

$$\frac{(P[a \rightarrow s], ht, h, ch, mq) \rightarrow (P[a := s'], ht', h', ch', mq')}{(P[a \rightarrow C[s]], ht, h, ch, mq) \rightarrow (P[a := C[s']], ht', h', ch', mq')}$$

Figure 9: Transition relation for statements, part 3.

$$\begin{array}{l}
\mathit{Init} = \\
\{(P, ht, h, ch, mq) \in \mathit{State} \mid \\
\ \ \exists\ a_p \in \mathit{ProcessAddress}, \\
\quad a_r \in \mathit{Address} \setminus \mathit{ProcessAddress}, \\
\quad a_s \in \mathit{Address} \setminus \mathit{ProcessAddress}. \\
\quad a_r \neq a_s \wedge \\
\quad P = f_0[a_p := a_p.\texttt{main}()] \wedge \\
\quad ht = f_0[a_p := \texttt{Process}, a_r := \texttt{Sequence}, a_s := \texttt{Sequence}] \wedge \\
\quad h = f_0[a_p := ha] \wedge \\
\quad ch = (\lambda(a_1, a_2) \in \mathit{ProcessAddress} \times \mathit{ProcessAddress}.\ \langle\rangle) \wedge \\
\quad mq = (\lambda a \in \mathit{ProcessAddress}.\ \langle\rangle) \\
\quad \text{where } ha = f_0[a_p := o_p, a_r := \langle\rangle, a_s := \langle\rangle] \\
\qquad\qquad o_p = f_0[\texttt{received} := a_r, \texttt{sent} := a_s]\}
\end{array}$$

Figure 10: Initial states.