The Semantics of ParalleX, v1.0

Matteo Cimini, Jeremy G. Siek, and Thomas Sterling

Indiana University

1. Introduction

This document provides a mathematically precise definition of the ParalleX execution model for large-scale parallel computing systems. The definition takes the form of an *operational semantics* (Felleisen & Hieb, 1992; Kahn, 1987; Landin, 1964; Plotkin, 2004). As such, the definition a) specifies the structure of an executing ParalleX system, that is a *snapshot* of the system (aka. *state* of the system), and b) provides a set of rules that determines the *transition* of one snapshot to another snapshot. The transition rules typically involve the parts of the ParalleX system devoted to *control*, e.g., program counters and analogous constructs. The entire execution of a ParalleX system is described by a sequence of transitions, starting with an initial snapshot S_1 and ending with the final snapshot S_n for which no more transition rules apply.

$$S_1 \longmapsto S_2 \longmapsto S_3 \longmapsto \cdots \longmapsto S_n$$

ParalleX describes parallel systems and their execution behavior, so it may seem odd to use a *sequence* of snapshots to capture the intended behavior. For example, suppose there are two computations, A and B, which could execute at the same time in snapshot S_2. If the two computations do not manipulate the same part of the snapshot, then executing the two computations in sequence produces the same behavior as having them run in parallel, that is, both orderings would result in the same snapshot S_4 as in (2) and (3) below. In the following we label the transitions with A or B to indicate which computation is executing when, and we write A|B to mean that A and B occur at the same time.

$$S_2 \longmapsto^{A|B} S_4, \tag{1}$$
$$S_2 \longmapsto^{A} S_3 \longmapsto^{B} S_4, \tag{2}$$
$$S_2 \longmapsto^{B} S_3' \longmapsto^{A} S_4. \tag{3}$$

If, on the other hand, the two computations do manipulate the same part of the snapshot, then there would be a *data race*, which in many systems leads to undetermined behavior. In ParalleX, data races are ruled out *by construction*, that is, by the combination of synchronization constructs and a programming discipline that is backed by automatic static checking. So in this example, the programmer would need to insert some kind of synchronization that would prevent computations A and B from executing at the same time, and the semantics would allow either ordering as shown below, where S_4may differ from S_4'.

$$S_2 \longmapsto^{A} S_3 \longmapsto^{B} S_4, \tag{4}$$

$$S_2 \longmapsto^{B} S_3' \longmapsto^{A} S_4'. \tag{5}$$

In this case, the resulting snapshot may depend on the ordering (S_4 or S_4') so the program may exhibit nondeterministic behavior. In such situations, the programmer is expected to limit the differences in behavior to ones that do not affect the overall correctness of the program.

The operational semantics described in this document specifies, for each ParalleX program, the set of all allowed transition sequences. Section 2 introduces the notation used throughout the document. The formal definition of ParalleX spans Sections 3, 4 and 5. In particular, Section 3 defines the structure of a ParalleX **Snapshot**. Section 4 defines ParalleX **Control** and Section 5 defines the **Transitions**. Section 6 defines the static **Discipline**. The discipline supports automatic static checking of ParalleX programs that guarantees data-race freedom.

This document covers the core entities and actions of ParalleX. In particular, it describes processes, complexes, parcels, and local control objects. With respect to actions, it specifies:

- Instantiation of new entities;
- Creation of new complexes;
- Parcel delivery;
- Memory operations (allocate, load, store, free, and copy);
- Arithmetic operations;
- Control of the execution flow (if, repeat, and subroutines);
- Arrays;
- Migration of entities and data to a different locality.

The future work relevant to this document includes:

- Expanding the formal semantics to the entirety of ParalleX, which includes the addition of hierarchical names, hardware resources, and percolation.
- Completing the static checking discipline of Section 6.
- Expand this document in concert with ParalleX as it grows to include features to address reliability, energy, real-time constraints, and introspection.

2. Preliminaries on Mathematical Notations

Before describing the ParalleX execution model, the mathematical entities and notation used in the description are described. Numbers, symbols, tuples, sets, multisets, and maps (i.e. functions). A **symbol** is a string of characters. A **tuple** is a sequence of comma-separated entities enclosed in angle brackets, such as $\langle 4,5,3,5\rangle$. A **set** is a collection of unique entities in which the ordering does not matter, such as $\{4,3,5\}$. Set membership is written with $\in$, so $3 \in \{4,3,5\}$ is true and $7 \in \{4,3,5\}$ is false. A **multiset** is like a set but allows multiple copies of an item. Entities can be arbitrarily nested within entities. For example, $\langle 5,\{2,3\},7\rangle$ is a tuple containing three entities. The second entity is a set that itself contains two entities: the numbers 2 and 3. Standard operations are used for sets such as the union of sets '$\cup$' and set difference '$-$'. Set difference on a multiset removes only one instance of the elements being removed, i.e. $\in \{3,4,3,5\}$ - $\{3\}$ results in the multiset $\{3,4,5\}$.

A **map** associates an output value with each of its inputs. The traditional notation $\{0 \mapsto 0, 1 \mapsto 1, 2 \mapsto 4, 3 \mapsto 9\}$ to describe maps is used. This particular map associates the numbers 0 through 3 with their square. The domain (or **dom** for short) of a map is the set of all input elements. The domain of this map is $\{0,1,2,3\}$.

Meta-variables are used for the purpose of referring to mathematical entities. They are written in italics. Meta-variables are not themselves entities. For example, the meta-variable 'T' is defined to refer to the set $\{2,3\}$ with an equation such as

$$T = \{2,3\}.$$

When a meta-variable is used in an expression, such as using T in the expression $3 \in T$, the expression has the same meaning as if the meta-variable T were replaced by its definition. In this case, $3 \in T$ has the same meaning as $3 \in \{2,3\}$. As another example, the expression $\langle 5,T,7\rangle$ refers to the same entity as $\langle 5,\{2,3\},7\rangle$.

Meta-variables are particularly useful for large entities. For example, the following equation define the meta-variable F to be the above mapping from integers to their squares.

$$F = \{0 \mapsto 0, 1 \mapsto 1, 2 \mapsto 4, 3 \mapsto 9, 4 \mapsto 16, \dots\}$$

Now F can be used succinctly to talk about applying this map to several different inputs.

$$F(1) = 1, \quad F(3) = 9$$

Equations are used in two different ways: the first case defines the meta-variable F and the second case tests for sameness, that is, the equation $F(1) = 1$ is true because $F(1)$ is the same entity as 1. Meta-variables are often used to store some result and use it elsewhere, for example $X = 4$ and $Y = 16$ in the expressions

$$F(2) = X \text{ and } F(X) = Y.$$

In this document, we often construct functions that are similar to already-constructed functions except that they differ on one input. The notation $F\{in \mapsto out\}$ refers to the map that is just like some mapping F except that it maps *in* to *out*.

For example, when $F = \{0 \mapsto 0, 1 \mapsto 1, 2 \mapsto 4, 3 \mapsto 9, \ldots\}$ as above, we have the map

$$F\{2 \mapsto 0\} = \{0 \mapsto 0, 1 \mapsto 1, 2 \mapsto 0, 3 \mapsto 9\}.$$

Records (i.e., structures) are modeled as maps whose domains are sets of symbols that represent the field names of the record. For instance, a record R with two fields field_1 and field_2 with values 3 and 4, respectively, is represented as the map $R = \{\mathrm{field}_1 \mapsto 3, \mathrm{field}_2 \mapsto 4\}$. As records are maps, they can be read-from and updated using map notation. For example, $R(\mathrm{field}_1)$ reads field_1 of the record R and $R\{\mathrm{field}_1 \mapsto 5\}$ updates field_1 of the record to 5. More precisely, we have

$$R(\mathrm{field}_1) = 3$$
$$R\{\mathrm{field}_1 \mapsto 5\} = \{\mathrm{field}_1 \mapsto 3, \mathrm{field}_2 \mapsto 5\}.$$

3.1 Snapshot

A ***snapshot*** is a record that consists of a map M, which represents the memory of the snapshot, and a multiset of parcels Q.

$$S = \{\text{memory} \mapsto M, \text{parcels} \mapsto Q\}$$

The map M is the abstraction of an unbounded number of memory cells that are indexed by virtual addresses (addresses for short). Memory cells come in two flavors, ***mutable cells*** that may be written-to multiple times and ***register cells*** which may be written-to only a single time and for which reads block until the first write (Arvind, Nikhil, & Pingali, 1987; Friedman & Wise, 1979; Halstead Jr., 1985). The content of a memory cell is denoted with

$$M(a) = v_{\ell}^{m,p}.$$

The address a maps to the cell containing the value v which resides in locality ℓ, has multiplicity m, and is owned by the process at address p. Each ParalleX system has a finite number of ***localities*** that roughly correspond to compute nodes. This document does not consider physical addresses, as they are below the desired level of abstraction. The ***multiplicity*** is either the symbol *, to indicate a mutable cell or 1 to indicate a register cell. Formally, the memory M map each virtual address to a 4-tuple $\langle v, \ell, m, p \rangle$ that contains the value in the cell (symbol v), its locality (symbol l), multiplicity (symbol m), and its owning process (symbol p). When a register has not yet been written-to, its content is *empty*, written ■. Otherwise the memory cell contains a value. The description of *values* is postponed to the next section.

Graphically: A memory is represented in the following way. The memory below contains two integers at addresses 1001 and 1002 and a real number at 1003, while the cell memory at 1004 has not been written-to yet. The values at addresses 1001 and 1003 are at the same locality ℓ_1 and are mutable cells. The integer at address 1002 is at a different location ℓ_2 and is a register cell. The dots denote that the memory can contain other entries.

Memory:

1001	$34_{\ell 1,*}^{1005}$
1002	$45_{\ell 2,1}^{1008}$
1003	$3.1415_{\ell 1,*}^{1015}$
1004	■

• • •

A ***parcel*** is the mechanism by which data action and control is transmitted between localities. A parcel is closely related to the notion of active message. Parcels are not first class entities and do not have addresses. Parcels reside in the multiset Q that contains all of the parcels that are in flight. A parcel consists of the address a of the receiver, a method name *mname*, payload data item *d,* and the continuation *cont*.

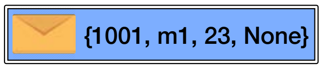
(The payload contains the arguments for the method invocation.) A parcel π takes the following form.

$$\pi = \{\text{destination} \mapsto a, \text{method} \mapsto mname, \text{payload} \mapsto d, \text{continuation} \mapsto cont\,\}$$

A **continuation** is either the symbol None, denoting no continuation, or a mapping from labels to parcels. Labels are names and can distinguish the role of the action being taken by the continuations. For example, the label 'error' can be associated to the parcel to send when an error occurs. Similarly, the label 'next' can be associated to what to do next.
The description of *data items* is postponed to the next section.

Graphically: Below is the graphical representation of a parcel that is destined to address 1001 for the invocation of the method *m1* with argument 23. Notice that such parcel has no continuation, hence it carries the symbol None.

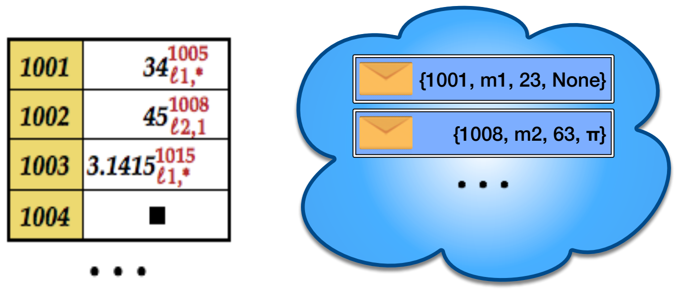
Parcel:



An example of snapshot.

The snapshot of a system is represented as follows, with the memory and the multiset of parcels together. The multiset of parcels is a cloud that can contains multiple parcels.

Snapshot:



3.2 Values (first-class entities)

A ***value*** is a first-class entity in that it has a virtual address and resides in the memory. A value can be a *data item*, a *compute complex* or an *object.*

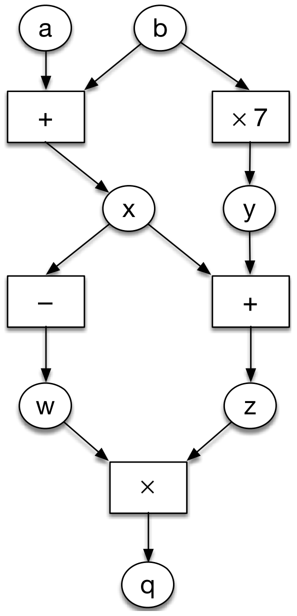
A ***data item*** is either atomic or compound. Atomic data includes integers, floating-point numbers, and virtual addresses. Compound data span multiple addresses, often contiguously addressed, and include entities such as arrays and tuples. Besides residing in the memory, data items can be placed at the payload of parcels.

A ***compute complex*** (complex for short) is the main computational entity in ParalleX. It is analogous to a thread except that a complex provides explicit support for internal fine-grained parallelism. A thread has a single program counter and executes a sequence of instructions. In contrast, a complex contains multiple dataflow graphs (Dennis, 1974), each of which may execute multiple instructions in parallel. Furthermore, a traditional thread has a procedure call stack (consisting of *activation frames*) and the only active subroutine is the one at the top of the stack. All other subroutines are essentially blocked, waiting for the return value from a call. In contrast, a ParalleX subroutine call is non-blocking, so the caller continues executing in parallel with the callee. As such, it does not make sense to use a stack per se to organize and control the subroutine calls.

A compute complex is a multiset of ***activation frames***, each of which is a structure equipped with *control*, *locals* and a *continuation*.

$$f = \{\text{control} \mapsto G, \text{locals} \mapsto L, \text{continuation} \mapsto cont\}$$

Dataflow Graph:

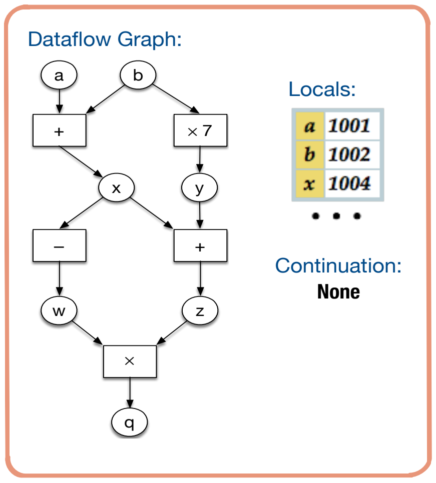


The control of a frame consists of a ***dataflow graph*** G. Graphically, a dataflow graph is represented as usual, as the example of the left shows. In Parallex a dataflow graph is a multiset of instruction nodes. (The instructions are defined in the Section Control.) For example, the dataflow graph on the left is the following multiset of instructions:

$$\{x \leftarrow a + b, y \leftarrow b \times 7, z \leftarrow x + y, w \leftarrow -x, q \leftarrow w \times z\}$$

The variables mentioned in the dataflow graph are divided in inputs and outputs. For example, in the instruction $x \leftarrow a + b$ the variables a and b are inputs and x is an output. The edges of the dataflow graph are implicit: there is an edge from node n_1 to node n_2 if one of the outputs of n_1 is an input of n_2.

Frame:

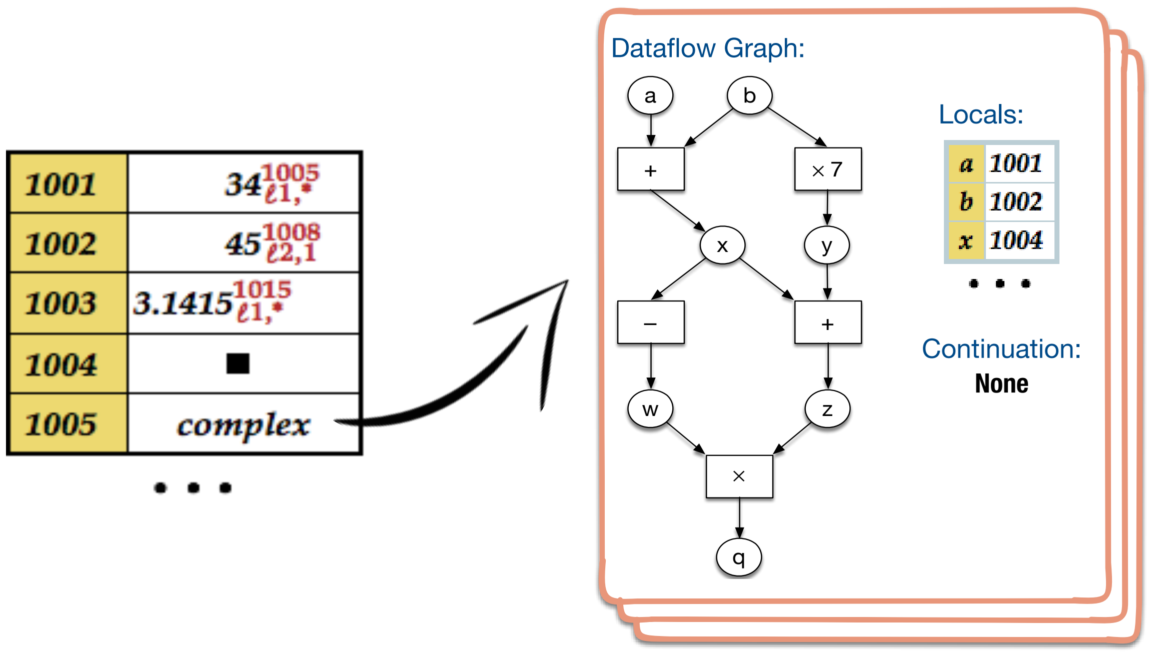


Instructions mention variables. These are the locals of a dataflow graph. The values of locals are stored by a map L from variable names to addresses of registers in the memory. Therefore, L does *not* contain the actual data for variables but it contains the virtual addresses where the actual data is stored. The picture on the left shows how frames are graphically represented.

An example of complex in the memory

The memory below contains a complex at address 1005, which has been expanded on the right. The complex contains several frames. This is pictorially displayed as if the complex is made of several pages, each of which is a frame. The frame at the top is the one being considered for computation. In this case, the dataflow graph strives to perform $x \leftarrow a + b$ and $y \leftarrow b \times 7$. As the locals a and b point to memory cells that have a value set, these instructions can take place and we say that the frame is *active*, and so is the complex. These instructions can be performed in parallel. It is importat to notice that all the other instructions are blocked. For example, $w \leftarrow -x$ cannot immediately take place because x has not received a value yet.

Complex:



A ParalleX ***object*** is an abstraction that encapsulates a method table MT, which maps method names to method descriptions (defined in the Control section), and a field table FT, which maps names to data items.

$$obj = \{\text{methods} \mapsto MT, \text{fields} \mapsto FT, \text{kind} \mapsto k\}$$

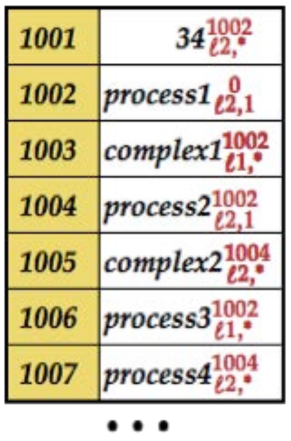
In ParalleX there are two types of objects: *processes* and *local control objects* (*LCOs*). The content of the field 'kind' differentiates processes from LCOs: for processes the field 'kind' is set to the symbol 'Process' and for LCOs is set to the symbol 'LCO'.

A ParalleX ***process*** is an object that can execute multiple compute complexes at the same time, can have data and spawn subprocesses. Processes also provide memory protection, as discussed later.

Processes are logical contexts

The memory cell at the virtual address of a process contains a structure obj, that is, the description of the object. However, processes can have data and spawn computational entities such as other processes or complexes. An object is virtually a logical context that contains its data and tasks. The logical context is not explicit but can be recovered through the parenthood information in the memory of the system. Below we show an example. *process1* has some data, one complex (*complex1*) and two subprocesses: *process2* and *process3*. *process3* itself contains the subprocess *process4*. The parent address of the initial process *process1* is 0 by convention.

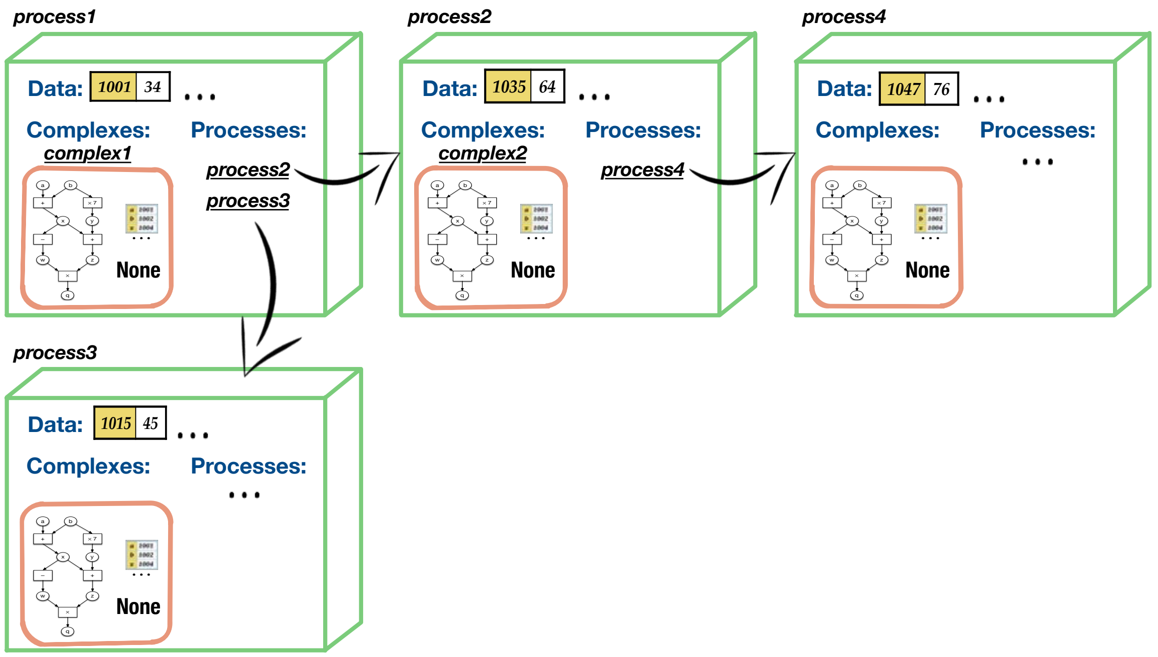
Memory layout:



1001	$34^{1002}_{\ell 2,\bullet}$
1002	$\mathit{process1}^{0}_{\ell 2,1}$
1003	$\mathit{complex1}^{1002}_{\ell 1,\bullet}$
1004	$\mathit{process2}^{1002}_{\ell 2,1}$
1005	$\mathit{complex2}^{1004}_{\ell 2,\bullet}$
1006	$\mathit{process3}^{1002}_{\ell 1,\bullet}$
1007	$\mathit{process4}^{1004}_{\ell 2,\bullet}$

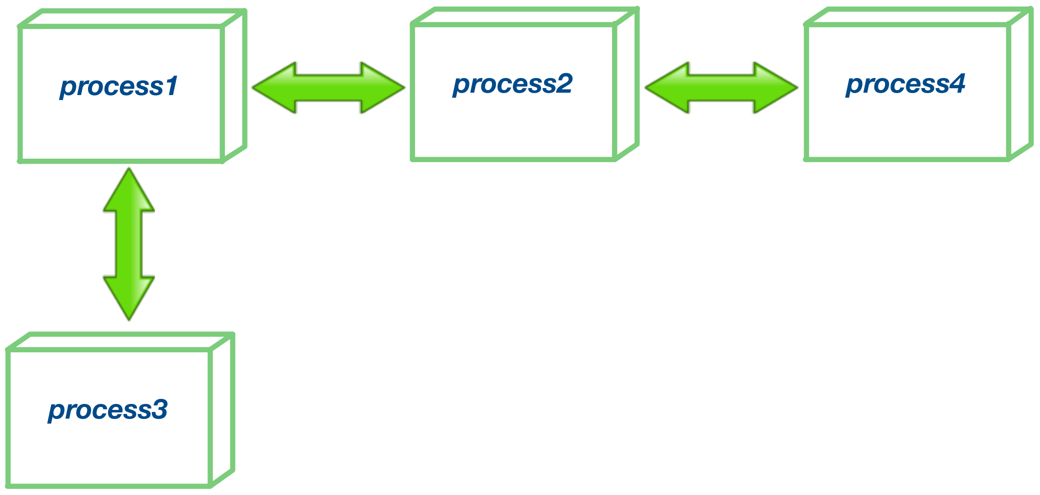
• • •

Logical contexts made explicit



Memory protection

Processes provide memory protection. A process may access the memory created by any of its ancestors or descendents. For the previous example, the is depicted as follows, where the green arrows denote that the pointed processes have read/write access to each other's data. Below, *process1* and *process4* also can communicate but, for example, *process3* cannot communicate with *process2* nor *process3*.



Processes can span multiple Localities

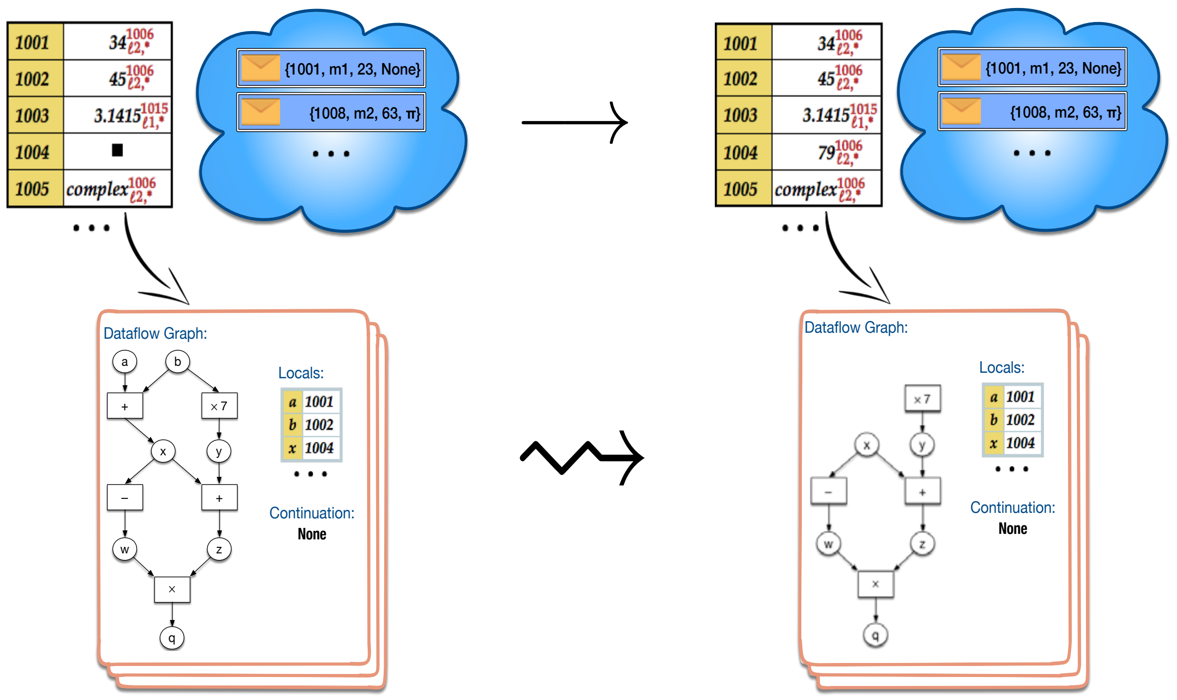
A process may span multiple localities in that its data, processes and complexes may be located in different localities. The locality associated with the virtual address of the process itself is its *primary* locality. As an example, the primary locality of *process1* is ℓ_2. The data at address 1001 and *process2* are part of the process and are in the same locality. While *complex1* and *process2* are part of *process1* they are localted locality ℓ_1, most probably because they have moved to another compute node for energy-efficiency reasons. Therefore *process1* spans the localities ℓ_1 and ℓ_2 where the latter is the primary locality of *process1*.

A ***local control object*** (LCO) is an object that is *synchronous* in that it can have at most one method running at a time. The notion of LCO is closely related to the notion of a monitor. An LCO has only one complex that resides at the same locality of the LCO itself. An LCO is therefore located in precisely one locality.

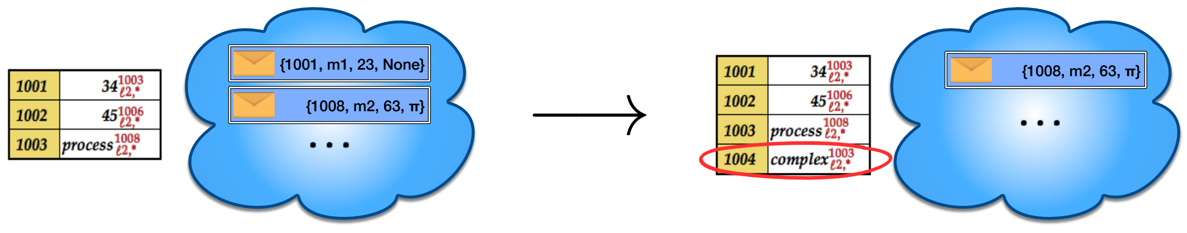
A summary of how computation evolves

At any time one of the active complexes can be selected for execution. Within the complex, an activation frame is selected. Each frame has a dataflow graph containing multiple instructions and one is chosen among those that are *ready*, that is, all inputs to the instruction have been written-to. Depending on which instruction is chosen, the execution may update the memory or change the multiset of parcels. The picture below shows the transition for an addition operation. The snapshot is on the left. The execution of instruction $x \leftarrow a + b$ updates the memory with $x =$

79. The instruction itself is consumed after execution. Finally, address 1005 contains the resulting (modified) complex.



At any time a parcel in the multiset can be chosen for delivery. In the previous example, the system could have chosen the parcel directed to 1001. Parcels are actions at a distance: they trigger the creation of a complex in a different locality. (More kinds of actions will be added as this document becomes more complete.) The example below shows the transition that derives from delivering a parcel. The snapshot contains a parcel destined to 1001. This parcel causes the execution of method $m1$ with its parameter initialized to 23. The parent process of the integer 34 is *process* at address 1003. It is *process* that contains the method table and therefore the dataflow graph for $m1$. The new complex is placed in a new address 1004 and has the same locality and parent as 1001. The multiplicity for complexes is always *.



4. Control

The control of a ParalleX system is specified in a format called *ParalleX Bytecode Representation* (PBR). PBR, defined in Figure 4, is not meant for direct programming by humans, but should be thought of as the target of compilation from higher-level languages, similar to the role of Java Bytecode with respect to the Java language.

ParalleX Spec.	*spec* ::=	$\{name_1 \mapsto proc_1, \dots, name_n \mapsto proc_n\}$
Procedure Spec.	*proc* ::=	$\{\text{methods} \mapsto MT, \text{fields} \mapsto VS, \text{kind} \mapsto k\}$
Procedure Kind	*k* ::=	$\text{Process} \mid \text{LCO}$
Variable Spec.	*VS* ::=	$\{var_1 \mapsto T_1, \dots, var_n \mapsto T_n\}$
Method Table	*MT* ::=	$\{name_1 \mapsto ms_1, \dots, name_n \mapsto ms_n\}$
Method Spec.	*ms* ::=	$\{\text{params} \mapsto VS, \text{body} \mapsto G\}$
DataFlow Graph	*G* ::=	$\{instr_1, \dots, instr_n\}$
Instructions	*instr* ::=	$var_1, \dots, var_n \leftarrow op(var_1, \dots, var_k)$
Operations	*op* ::=	$\textbf{instantiate} \mid \textbf{invokeMethod}[name] \mid \textbf{allocate}[\mathrm{m}] \mid$ $\textbf{load} \mid \textbf{store} \mid \textbf{free} \mid \textbf{addressof} \mid \textbf{copy} \mid$ $+ \mid - \mid \times \mid \div \mid \textbf{const}[literal] \mid \textbf{if}[G_1, G_2] \mid$ $\textbf{repeat}[G_1, G_2, var_1, var_2, i] \mid \textbf{call}[sname] \mid$ $\textbf{array}[T, m] \mid \textbf{readAt} \mid \textbf{writeAt} \mid \textbf{continue}[label] \mid$ $\textbf{createContinuation}[label, mname] \mid$ $\textbf{addContinuation}[label, mname]$
Multiplicity	*m* ::=	$1 \mid *$

Figure 4. ParalleX Bytecode Representation

A ParalleX ***specification*** is a mapping of names to procedure specifications, called ***procedures***. (A procedure is analogous to a *class* in object-oriented systems.) One of the procedures must be named *main* and is implicitly launched when the program is run. This will be the initial process of the program. Each procedure has a set of ***fields*** whose lifetime is same as that of the running process and whose scope includes all of the methods of the procedure. A procedure includes a method table that contains one or more method specifications. One of the method specifications must be named *main*. The *main* method is implicitly invoked when the process is created. This will trigger the initial complex of a process.

A ***method specification*** contains a list of parameters and their types; this is the set of local variable declarations. Additionally, a method specification contains the body of the method, which is a dataflow graph.

5. Transitions

Further on Notation

When we access a memory cell as in $M(a) = v_{\ell}^{m,p}$, it is some times the case that not each of the information about location, multiplicity, and owner matters. In those cases, we simply omit the irrelevant information as in $M(a) = v$ or $M(a) = v_{\ell}$. When updating the content of a memory cell, if the multiplicity, owner, or locality is not specified then those attributes remain as they were.
For example, given $M(a) = 34_{\ell}^{m,p}$, the update $M\{a \mapsto 45\}$ is such that $M(a) = 45_{\ell}^{m,p}$.

The representation of some ParalleX entities may nest complex structures. It is typical for the formal semantics to work with maps inside maps. Map composition is used for looking up some nested value concisely. For example, the formal semantics often uses the expression $f(\text{locals})(x)$ that first calculates $f(\text{locals})$ for extracts the map L of the frame f and then calls $L(x)$. Another recurrent example of composition is

$$M(a)(\text{methods})(mname) = \{\text{params} \mapsto \{x_1 \mapsto T_1, \dots, x_n \mapsto T_n\}, \text{body} \mapsto G\}$$

This expression retrieves the method specification associated with a name $mname$ within the method table of an object that is at address a.

Again, $M(a)$ returns an object o that, whether process or LCO, it contains (at least) $\{\text{methods} \mapsto MT, \text{fields} \mapsto FT\}$. The composition $M(a)(\text{methods})$ returns the method table MT from which the method $mname$ is then looked up.

Auxiliary Definitions

The locals of an activation frame do not contain the actual data to which a dataflow graph refers. Rather, they contain the address where this data can be found in the memory. For convenience, the formal semantics uses shorthand for hiding the complexity behind this indirection. The auxiliary function *lookup* is used to lookup some data. Given a name x, a frame f and a memory M, we first lookup the address in the locals of the frame f and then query the memory M for returning the actual value. We define the following shorthand.

$$lookup(x, f, M) = M(a) \quad \text{where } \; a = f(\text{locals})(x)$$

The function *update* hides the same indirection when updating the data associated with a local variable. This function takes a map $\{x_1 \mapsto d_1, \dots, x_n \mapsto d_n\}$ of new assignments, a frame f and a memory M, updates M at the addresses pointed to by xs, and assigns the corresponding ds.

$$update(\{x_1 \mapsto d_1, \dots, x_n \mapsto d_n\}, f, M) = M\{a_1 \mapsto d_1, \dots, a_n \mapsto d_n\}$$
$$\text{where } a_1 = f(\text{locals})(x_1), \dots, a_n = f(\text{locals})(x_n)$$

We also define a family of functions for collecting the local variables that are used in a dataflow graph.

$$locals\_of(z \leftarrow x + y) = \{x, y, z\}$$
$$locals\_of(G) = \{x \mid \exists instr.\, x \in localsOf(instr) \; and \; instr \in G\}$$

The auxiliary function *distinct* takes two or more sets of addresses as arguments and makes sure that the sets do not contain the same address, thereby ensuring that these "new" addresses are really new.

The next auxiliary function is responsible for allocating the registers associated with the local variables of a compute complex. This function returns the map for the local variables and the updated memory.

$$allocate\_locals(G, p, \ell, M) = \langle\{x_1 \mapsto a_1, \dots, x_n \mapsto a_n\}, M\{a_1 \mapsto \blacksquare_\ell^{1,p}, \dots, a_n \mapsto \blacksquare_\ell^{1,p}\}\rangle$$

where $locals\_of(G) = \{x_1, \dots, x_n\}$ and $distinct(\{a_1\}, \dots, \{a_n\}, \text{dom}(M))$.

The function above returns a pair. The formal semantics makes use of the result of such function with the expression $allocate\_locals(G, p, \ell, M) = \langle L_{locals}, M' \rangle$. This is convenient to refer to the locals and the updated memory with symbols L_{locals} and M', respectively.
The allocation and initialization of variables (such as parameters of a method or subroutine) is specified as follows. The $x_1, \dots, x_n$ are the variables and $z_1, \dots, z_n$ are the initializers. Each variable is associated with a memory cell in locality ℓ.

$$allocate\_init(x_1, \dots, x_n, z_1, \dots, z_n, f, p, m, \ell, M) = \langle L, M' \rangle$$

where $L = \{\, x_1 \mapsto a_1, \dots, x_n \mapsto a_n\}$
and $M' = M\{a_1 \mapsto lookup(z_1, f, M)_\ell^{m,p}, \dots, a_n \mapsto lookup(z_n, f, M)_\ell^{m,p}\}$
and $distinct(\{a_1\}, \dots, \{a_n\}, \mathrm{dom}(M))$.

When $allocate\_init$ is called with data instead of variables, memory lookup is skipped, as follows.

$$allocate\_init(x_1, \dots, x_n, d_1, \dots, d_n, f, p, m, \ell, M) = \langle L, M' \rangle$$

where $L = \{\, x_1 \mapsto a_1, \dots, x_n \mapsto a_n\}$
and $M' = M\{a_1 \mapsto d_1, \dots, a_n \mapsto d_n\}$
and $distinct(\{a_1\}, \dots, \{a_n\}, \mathrm{dom}(M))$.

As LCOs can execute only one complex at any given time, the $available$ function checks whether an LCO at address a_p does not contain any complex. This function is called at any invoke operation destined to an LCO.
$available(a_p) = true$
  whenever $M(a_p)(\mathrm{kind}) = \mathrm{LCO}$ implies $\neg\exists a', C, m, \ell.\, a' \neq a$ and $M(a) = C_\ell^{m,a_p}$
  ($false$, otherwise)

Instructions of a process may access memory cells created by another process. This access is granted according to a specific discipline. In particular, a process can access the memory cells allocated by itself or that belongs to any of its ancestors and sub-process. The *ancestors* function calculates the ancestors of a given address.

$$ancestors\ (a, M) = \{a\} \cup ancestors\ (a_p, M) \qquad \text{if } M(a) = v^{a_p}$$

The function $access\_control$ checks whether the process at address a_1 has the right to access the address a_2.
$access\_control(a_1, a_2) = true$ whenever
  $a_2 \in \mathrm{ancestors}(a_1)$ or $a_1 \in \mathrm{ancestors}(a_2)$
  ($false$, otherwise)

ParalleX selects instructions for execution. However, instructions are nested deep inside maps (within complexes, within frames and finally within dataflow graphs). The formal semantics uses the function $fetch\_instruction(M) = (instr, {f'}_\ell^{m,a_p}, M')$ for conveniently fetching an instruction. This function acts on the memory M and returns the picked instruction $instr$, the frame ${f'}_\ell^{m,a_p}$ from which $instr$ has been selected and the new memory M' where the instruction is removed at the proper place.

$$fetch\_instruction(M) = (instr, {f'}_{\ell}^{m,a_p}, M')$$

where
$\exists a, C, m, \ell, a_p.\ M(a) = C_{\ell}^{m,a_p}$ and
$f \in C$, and $instr \in f$, and
$C' = C - f \cup \{f'\}$ and $f' = f - \{instr\}$.
$M' = M\{a \mapsto C'\}$

In what follows, we provide the formal rules that defines when a ParalleX snapshot can transition to another snapshot.

Instantiate an Object

The **instantiate** operation allocates a new object (process or LCO) in the same locality as the executing complex, sets up its method table, allocates and initializes its fields (from the arguments $y_1, \dots, y_n$), and invokes its *main* method (binding the arguments $z_1, \dots, z_k$ to its parameters). Formally, the transition rule can fire in some snapshot S if the following conditions are met:

- $M_0 = S(\text{memory}), fetch\_instruction(M_0) = (instr, f_{\ell}^{p'}, M_1)$
- $instr = y \leftarrow \textbf{instantiate}(pname, y_1, \dots, y_n, z_1, \dots, z_k)$
- $prog(pname) =$
  $\{\text{methods} \mapsto MT, \text{fields} \mapsto \{var_1 \mapsto {T'}_1, \dots, var_n \mapsto {T'}_n\}, \text{kind} \mapsto objkind\}$
- $MT(params) = \{x_1 \mapsto T_1, \dots, x_k \mapsto T_k\}$

The new snapshot *S'* is defined as follows:

$$S' = S(\text{memory} \mapsto update(y, a_p, f, M_1)\{\, a_p \mapsto obj_{\ell}^{1,p'}, a' \mapsto {C_{main}}_{\ell}^{1,a_p}\})$$

where

- $obj = \{\text{methods} \mapsto MT, \text{fields} \mapsto FT, \text{kind} \mapsto objkind\}$
- $distinct(\{a_p\}, \{a'\}, \text{dom}(M_1))$
- $C_{main} = \left\{\begin{Bmatrix} \text{control} \mapsto MT(\text{main})(\text{body}), \\ \text{locals} \mapsto \{\text{this} \mapsto a'\} \cup L_{params} \cup FT, \\ \text{continuation} \mapsto \text{None} \end{Bmatrix}\right\}$
- $allocate\_init\big(var_1, \dots, var_n, y_1, \dots, y_n, f, a_p, *, \ell, M_1\big) = \langle FT, M_2 \rangle$
- $allocate\_init\big(x_1, \dots, x_k, z_1, \dots, z_k, f, a_p, 1, \ell, M_2\big) = \langle L_{params}, M_3 \rangle$

Invoke a Method of an Object (Create a Complex)

The **invokeMethod** operation begins the execution of a method by creating a complex. The receiver of the invocation can be any addressable entity and the new complex is placed within the same locality as the receiver. The code for the method comes from the process that owns the receiver and the creation of a complex consists in forming a complex (a multiset of frames) with only one frame that executes that code. If the receiver is local, that is, if the locality of the sender and of the receiver coincide, then complex creation happens immediately. If the receiver is not local, then a parcel is sent to the receiver. The arguments to the method are passed by value, that is, the contents of the registers associated with the arguments

$z_1, \dots, z_n$. A ParalleX program that wishes to instead pass by reference can apply the **addressof** instruction to the arguments (not defined yet). The last argument of **invokeMethod** is the continuation for this operation.

Invoke Method on Local Receiver

The first transition rule handles the local case, when the executing complex and the receiver process are in the same locality. If the object owner of the receiver is an LCO then the transition requires that the LCO has no other active complexes. We allocate the new complex and registers for its local variables, and we pass (by reference) the parameters. Formally, this transition rule can fire in some snapshot S if the following conditions are met:

- $M_0 = S(\text{memory}), \mathit{fetch\_instruction}(M) = (\mathit{instr}, f_{\ell_1}, M_1)$
- $\mathit{instr} = \textbf{invokeMethod}[\mathit{mname}](r, z_1, \dots, z_n, y)$
- $a_r = \mathit{lookup}(r, f, M_1)$
- $M_1(a_r) = v_{\ell_2}^{m,a_p}, \ell_1 = \ell_2$
- $\mathit{available}(a_p)$
- $M_1(a_p)(\text{methods})(\mathit{mname}) = \{\text{params} \mapsto \{x_1 \mapsto T_1, \dots, x_n \mapsto T_n\}, \text{body} \mapsto G\}$
- $\mathit{cont} = \mathit{lookup}(y, f, M_1)$

The new snapshot S' is defined as follows:

$$S' = S\{\text{memory} \mapsto M_3\{a_{new} \mapsto C_{\ell_2}^{1,a_p}\}\}$$

where

- $C = \left\{\begin{Bmatrix} \text{control} \mapsto G, \\ \text{locals} \mapsto (\{\text{this} \mapsto a_r\} \cup L_{args} \cup L_{locals}), \\ \text{continuation} \mapsto \mathit{cont} \end{Bmatrix}\right\}$
- $\mathit{allocate\_locals}(G, a_p, \ell_2, M_1) = \langle L_{locals}, M_2 \rangle$
- $\mathit{allocate\_init}(x_1, \dots, x_n, z_1, \dots, z_n, f, a_p, 1, \ell_2, M_2) = \langle L_{args}, M_3 \rangle$
- $a_{new} \notin \text{dom}(M_3)$

Initiate Invoke Method for Remote Receiver

The second transition rule handles the remote case, in which a parcel is created to transmit the method invocation to the receiver.
Formally, this transition rule can fire in some snapshot S if the following conditions are met:

- $M = S(\text{memory}), \mathit{fetch\_instruction}(M) = (\mathit{instr}, f_{\ell_1}, M')$
- $\mathit{instr} = \textbf{invokeMethod}[\mathit{mname}](r, z_1, \dots, z_n, y)$
- $a_r = \mathit{lookup}(r, f, M')$
- $M'(a_r) = v_{\ell_2}^{m,a_p}$,
- *either* $\ell_1 \neq \ell_2$ or $\ell_1 = \ell_2$ *and* not $\mathit{available}(a_p)$),
- $\mathit{cnt} = \mathit{lookup}(y, f, M')$

The new snapshot S' is defined as follows:

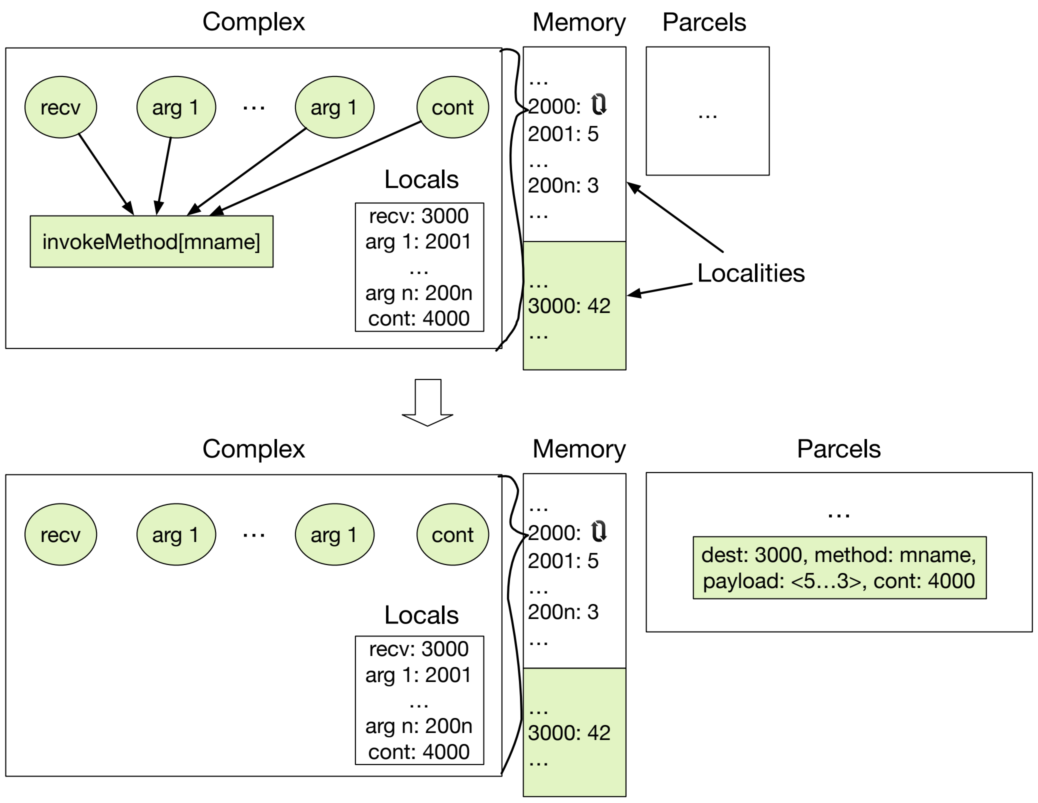
$$S' = S\{\text{memory} \mapsto M', \text{parcels} \mapsto Q \cup \{\pi\}\}$$

where

- $\pi = \{\text{destination} \mapsto a_r, \text{method} \mapsto \mathit{mname}, \text{payload} \mapsto d, \text{continuation} \mapsto \mathit{cnt}\}$

- $Q = S(\text{parcels})$ and $d = \langle lookup(z_1, f, M'), \ldots, lookup(z_n, f, M')\rangle$

The Figure below shows an example of a method invocation triggering the creation of a parcel to trigger the method on a remote receiver, that is, a receiver in a locality that is different from the complex that is invoking the method.



Parcel Delivery (Invoke Method)

At any time, the system can deliver one of the parcels that are sitting in the multiset of parcels (if other conditions are met). This transition rule can fire in some snapshot S if the following conditions are met:

- $M = S(\text{memory})$ and $Q = S(\text{parcels})$
- $\pi \in Q$
- $\pi = \{\text{destination} \mapsto a_r, \text{method} \mapsto mname, \text{payload} \mapsto \langle d_1, \ldots, d_n\rangle, \text{continuation} \mapsto cont\}$
- $M(a_r) = v_\ell^{m,a_p}$
- $available(a_p)$
- $M(a_p)(\text{methods})(mname) = \{\text{params} \mapsto PS, \text{body} \mapsto G\}$
- $PS = \{x_1 \mapsto T_1, \ldots, x_n \mapsto T_n\}$

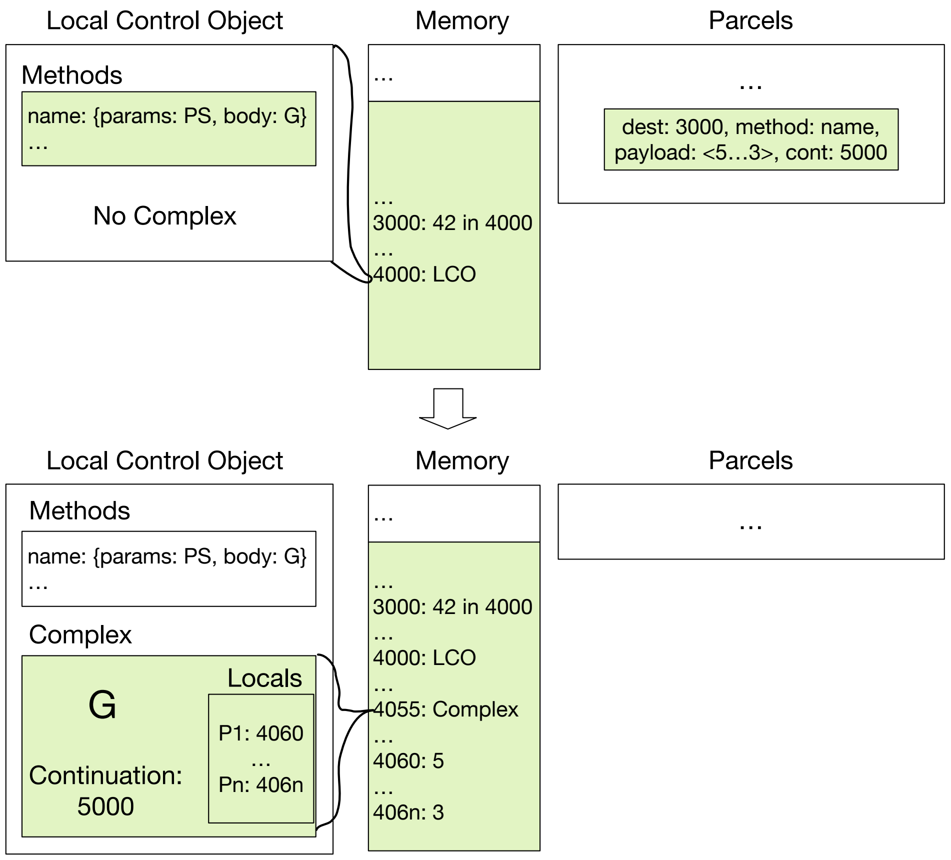
The new snapshot S' is defined as follows:

- $S' = S\{\text{memory} \mapsto M'\left\{a_{new} \mapsto C_\ell^{1,a_p}\right\}, \text{parcels} \mapsto Q - \{\pi\}\}$

where

- $C = \left\{ \left\{ \begin{array}{c} \text{control} \mapsto G, \\ \text{locals} \mapsto (\{\text{this} \mapsto a_r\} \cup L_{args} \cup L_{locals}), \\ \text{continuation} \mapsto cont \end{array} \right\} \right\}$
- $allocate\_locals(G, a_p, \ell_2, M) = \langle L_{locals}, M' \rangle$
- $L_{args} = \{x_1 \mapsto d_1, \dots, x_n \mapsto d_n\}$
- $a_{new} \notin \text{dom}(M')$

The Figure below shows an example of parcel delivery in the case when the receiver is an LCO. In this case, the LCO must not have an active complex prior to fetching the parcel and creating the new complex.



Continuation Creation

The **createContinuation** operation stores a continuation that prescribes action for one label. Actions for further labels can be specified by using this continuation with the **addContinuation** instruction.
The created continuation can then be passed as the last argument of an **invokeMethod** instruction.

- $M = S(\text{memory}), fetch\_instruction(M) = (instr, f, M')$
- $instr = \ x \leftarrow \textbf{createContinuation}[label, mname](r, z_1, \dots, z_n, y)$
- $a_r = lookup(r, f, M')$

The new snapshot S' is defined as follows:

- $S' = S\{\text{memory} \mapsto M''\{a_{new} \mapsto \{label \mapsto \pi\}\}\}$

where

- $\pi = \{\text{destination} \mapsto a_r, \text{method} \mapsto mname, \text{payload} \mapsto d, \text{continuation} \mapsto cnt\}$
- $cnt = lookup(y, f, M')$ and $d = \langle lookup(z_1, f, M'), \dots, lookup(z_n, f, M')\rangle$
- $a_{new} \notin \text{dom}(M')$
- $M'' = update(\{x \mapsto a_{new}\}, f, M')$

Adding to Continuations

The **addContinuation** operation augments a continuation with the association of some action to a label. The returned variable can then be passed as the last argument of an **invokeMethod** instruction.

- $M = S(\text{memory}), fetch\_instruction(M) = (instr, f, M')$
- $instr = z \leftarrow \textbf{addContinuation}[label, mname](x, r, z_1, \dots, z_n, y)$
- $cont_1 = lookup(x, f, M')$
- $a_r = lookup(r, f, M')$

The new snapshot S' is defined as follows:

- $S' = S\{\text{memory} \mapsto M''\{a_{new} \mapsto cont_1\{label \mapsto \pi\}\}\}$

where

- $\pi = \{\text{destination} \mapsto a_r, \text{method} \mapsto mname, \text{payload} \mapsto d, \text{continuation} \mapsto cont_2\}$
- $cont_2 = lookup(y, f, M')$ and $d = \langle lookup(z_1, f, M'), \dots, lookup(z_n, f, M')\rangle$
- $a_{new} \notin \text{dom}(M')$
- $M'' = update(\{z \mapsto a_{new}\}, f, M')$

Invoke the Continuation: local invocation

The **continue** operation invokes the continuation of the frame. If the invocation is local then it is immediately performed, otherwise a parcel is sent (next case).

- $M = S(\text{memory}), fetch\_instruction(M) = (instr, f_{\ell_1}, M_1)$
- $instr = \textbf{continue}[label]$
- $f = \{\text{control} \mapsto G, \text{locals} \mapsto L, \text{continuation} \mapsto cont\}$ and $\pi = cont(label)$
- $\pi = \{\text{destination} \mapsto a_r, \text{method} \mapsto mname, \text{payload} \mapsto d, \text{continuation} \mapsto cont_2\}$
- $d = \langle d_1, \dots, d_n \rangle$
- $M_1(a_r) = v_{\ell_2}^{m,a_p}, \ell_1 = \ell_2$
- $available(a_p)$
- $M_1(a_p)(\text{methods})(mname) = \{\text{params} \mapsto \{x_1 \mapsto T_1, \dots, x_n \mapsto T_n\}, \text{body} \mapsto G\}$

The new snapshot S' is defined as follows:

$$S' = S\{\text{memory} \mapsto M_3\{a_{new} \mapsto C_{\ell_2}^{1,a_p}\}\}$$

where

- $C = \left\{\begin{Bmatrix} \text{control} \mapsto G, \\ \text{locals} \mapsto (\{\text{this} \mapsto a_r\} \cup L_{args} \cup L_{locals}), \\ \text{continuation} \mapsto cont_2 \end{Bmatrix}\right\}$
- $allocate\_locals(G, a_p, \ell_2, M_1) = \langle L_{locals}, M_2\rangle$

- $allocate\_init(x_1, \dots, x_n, d_1, \dots, d_n, f, a_p, 1, \ell_2, M_2) = \langle L_{args}, M_3 \rangle$
- $a_{new} \notin \mathrm{dom}(M_3)$

Invoke the Continuation: remote invocation

The **continue** operation sends a parcel when the invocation is not local.
$M = S(\text{memory}), fetch\_instruction(M) = (instr, f_{\ell_1}, M')$

- $Q = S(\text{parcels})$
- $instr = \textbf{continue}[label]$
- $f = \{\text{control} \mapsto G, \text{locals} \mapsto L, \text{continuation} \mapsto cont\}$
- $\pi = cont(label)$
- $a_r = \pi(\text{destination})$
- $M'(a_r) = v_{\ell_2}^{m,a_p}$,
- $either\ \ell_1 \neq \ell_2$ or $\ell_1 = \ell_2$ and not $available(a_p)$).

The new snapshot S' is defined as follows:

$$S' = S\{\text{memory} \mapsto M', \text{parcels} \mapsto Q \cup \{\pi\}\}$$

Allocate a Memory Cell

The **allocate**$[T, m]$ operation creates a memory cell of type T at a new address, using multiplicity m, and places the cell in the same locality as the "hint" argument x, which is optional. If there is no hint argument, the locality is the same as the compute complex that executes the operation. The new address is returned into y.

This transition rule can fire in some snapshot S if the following conditions are met:

- $M = S(\text{memory}), fetch\_instruction(M) = (instr, f^{a_p}, M')$
- $instr = y \leftarrow \textbf{allocate}[T, m](x)$
- $a_x = lookup(x, f, M')$
- $M'(a_x) = v_\ell$
- $access\_control(a_p, a_x)$

The new snapshot S' is defined as follows:

$$S' = S\{\text{memory} \mapsto M''\{a_{new} \mapsto \blacksquare_\ell^{m,a_p}\}\}$$

where

- $a_{new} \notin \mathrm{dom}(M')$
- $M'' = update(\{y \mapsto a_{new}\}, f, M')$

Load from Memory

The **load** operation retrieves the values stored at the address in argument x, putting the loaded value into y. It also puts the address in z, which facilitates the static discipline described in the Discipline section.

This transition rule can fire in some snapshot S if the following conditions are met:

- $M = S(\text{memory}), fetch\_instruction(M) = (instr, f^{a_p}, M')$
- $instr = y, z \leftarrow \textbf{load}(x)$

- $a_x = lookup(x, f, M')$
- $M'(a_x) = d, access\_control(a_p, a_x)$

The new snapshot S' is defined as follows:

$$S' = S\{\text{memory} \mapsto update(\{y \mapsto d, z \mapsto a_x\}, f, M')\}$$

Store to Memory

The **store** operation places the value stored in *x* into the memory cell addressed by the address in *y*. It also puts the address of *y* in *z*, which facilitates the static discipline (see the Discipline section). If the multiplicity of the memory cell is 1 (write-one), then the memory cell must be empty (■) to perform the store.

This transition rule can fire in some snapshot *S* if the following conditions are met:

- $M = S(\text{memory}), fetch\_instruction(M) = (instr, f^{a_p}, M')$
- $instr = z \leftarrow \textbf{store}(x, y)$
- $d = lookup(x, f, M')$ and $a_y = lookup(y, f, M')$
- $M'(a_y) = v^m$ and $m = 1$ implies $v = ■$
- $access\_control(a_p, a_y)$

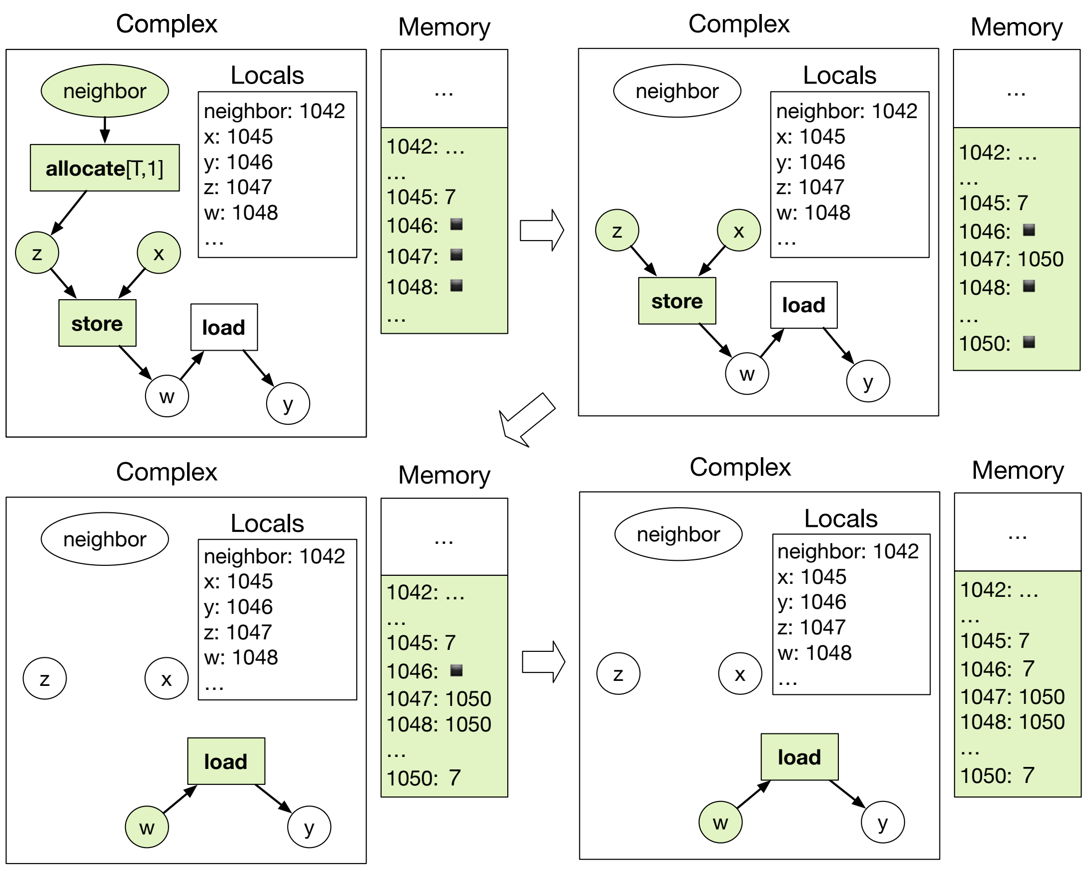
The new snapshot S' is defined as follows:

$$S' = S\{\text{memory} \mapsto M''\{a_y \mapsto d^m\}\}$$

where

- $M'' = update(\{z \mapsto a_y\}, f, M')$

The Figure below shows a sequence of operations: allocating a memory cell, storing the integer 7 into it, and then loading that integer back out again.



Free Memory

This transition rule can fire in some snapshot S if the following conditions are met:

- $M = S(\text{memory}), \mathit{fetch\_instruction}(M) = (\mathit{instr}, f^{a_p}, M')$
- $\mathit{instr} = \textbf{free}(x)$
- $a_x = \mathit{lookup}(x, f, M')$
- $\mathit{access\_control}(a_p, a_x)$

The new snapshot S' is defined as follows:

$$S' = S\{\text{memory} \mapsto (M' - \{a_x\})\}$$

Arithmetic Operations

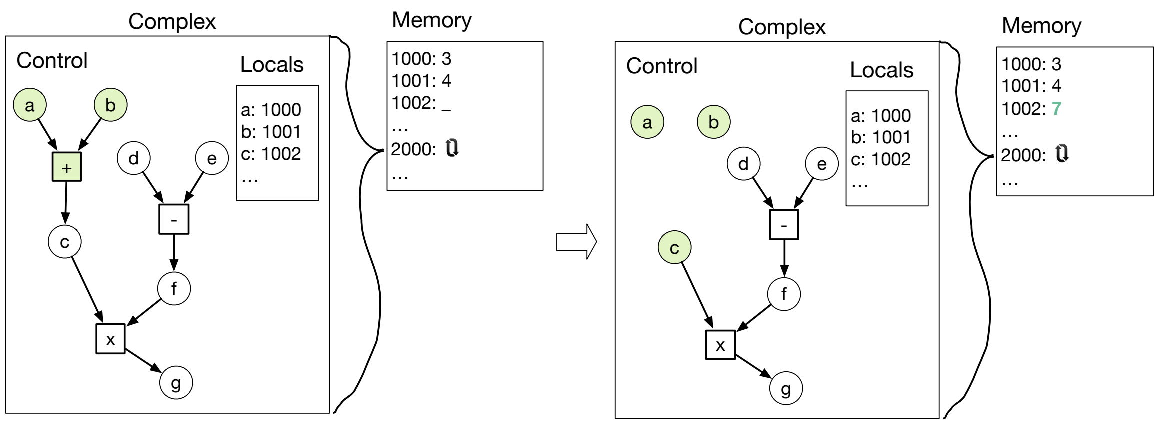
The following transition rule defines the behavior of the addition operation. The other arithmetic operations are similar. This transition rule can fire in some snapshot S if the following conditions are met:

- $M = S(\text{memory}), \mathit{fetch\_instruction}(M) = (\mathit{instr}, f, M')$
- $\mathit{instr} = z \leftarrow x + y$
- $d_x = \mathit{lookup}(x, f, M')$ and $d_y = \mathit{lookup}(y, f, M')$

The new snapshot S' is defined as follows:

$$S' = S\{\text{memory} \mapsto (\mathit{update}(\{z \mapsto d_x + d_y\}, f, M'))\}$$

The Figure below shows an example of this transition. The snapshot on the left has a ready addition operation in green. The snapshot on the right shows the results of the addition, with the result stored in the memory location for local variable c.



Constant Operations

This instruction stores a literal, such as '3' or 'True', into a variable.
This transition rule can fire in snapshot S if the following conditions are met:

- $M = S(\text{memory}), fetch\_instruction(M) = (instr, f, M')$
- $instr = x \leftarrow \mathbf{const}[lit]$

The new snapshot S' is defined as follows:

$$S' = S\{\text{memory} \mapsto update(x, lit, f, M')\}$$

Copy Operation

This transition rule can fire in snapshot S if the following conditions are met:

- $M = S(\text{memory}), fetch\_instruction(M) = (instr, f, M')$
- $instr = y \leftarrow \mathbf{copy}(x)$
- $lookup(x, f, M') = d_x$

The new snapshot S' is defined as follows:

$$S' = S\{\text{memory} \mapsto (update(y \mapsto d_x, f, M'))\}$$

AddressOf Operation

This transition rule can fire in snapshot S if the following conditions are met:

- $M = S(\text{memory}), M(a) = C, f \in C$
- $y \leftarrow \mathbf{addressOf}(x) \in f(\text{control})$
- $x \mapsto a' \in f(\text{locals})$

The new snapshot S' is defined as follows:

$$S' = S\{\text{memory} \mapsto (update(y \mapsto a', f, M))\{a \mapsto C'\}\}$$

where

- $C' = \{f'\} \cup (C - \{f\})$
- $f' = f\{\text{control} \mapsto (f(\text{control}) - y \leftarrow \mathbf{addressOf}(x))\}$

Conditional Operation

An **if** operation, of the form

$$\mathbf{if}[G_1, G_2](x)$$

activates either dataflow graph G_1 or G_2 depending on the value in x. It does so by augmenting the complex (that is, the multiset of frames) with an additional frame. As this operation modifies the multiset of frames of the selected complex, the function $fetch\_instruction$ is not adequate. Below, we fetch the complex with no auxiliary functions.

This transition rule can fire in snapshot S if the following conditions are met:

- $M = S(\text{memory}), M(a) = C, f \in C$
- $\textbf{if}[G_1, G_2](x) \in f(\text{control})$
- $lookup(x, f, M') = v_x$

If v_x = true then the output snapshot S' is the following:

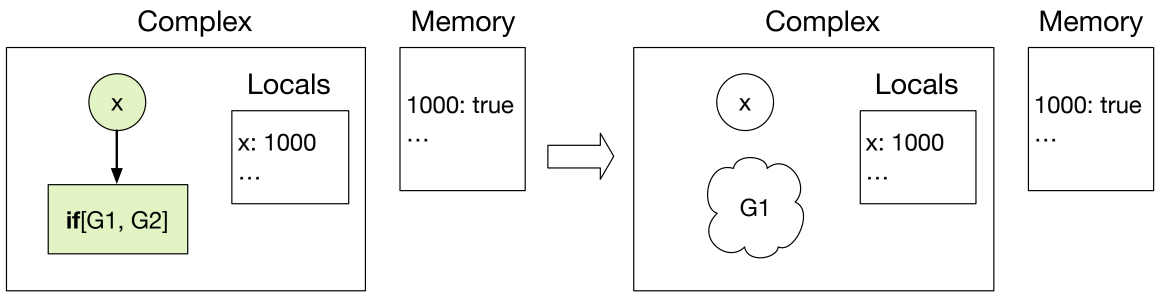
$$S' = S\{\text{memory} \mapsto M\{a \mapsto C'\}\}$$

where

- $C' = \{f'\} \cup (C - \{f\})$
- $f' = f\{\text{control} \mapsto (f(\text{control}) - \{\textbf{if}[G_1, G_2](x)\}) \cup G_1\}$

If v_x = false then the output snapshot S' is the same as above except that the graph G_2 is added to the control instead of G_1.

The Figure below depicts an example **if** operation. The condition is **true**, so the graph G1 (the cloud) is activated to become part of the current dataflow graph.



Repeat Operation

A **repeat** operation, of the form

$$\textbf{repeat}[G_C, G_B, y, x_1, \dots, x_n](z_1, \dots, z_n)$$

executes the body G_B as long as the condition G_C is true. The variables z_i are used to initialize the loop variables x_i. The condition G_C is expected to read from the x_i variables and put its Boolean result in variable y. The body of the loop G_B is expected to read from the x_i variables and put its results in x_i'. Once the condition is false, the loop discontinues. Each time through the loop, the graphs G_C and G_B become activated in a new frame.

This transition rule can fire in snapshot S if the following conditions are met:

- $M = S(\text{memory}), M(a) = C, f \in C$
- $\textbf{repeat}[G_C, G_B, y, x_1, \dots, x_n](z_1, \dots, z_n) \in f(\text{control})$
- $lookup(z_1, f, M) = d_1$ and …, and $lookup(z_n, f, M) = d_n$

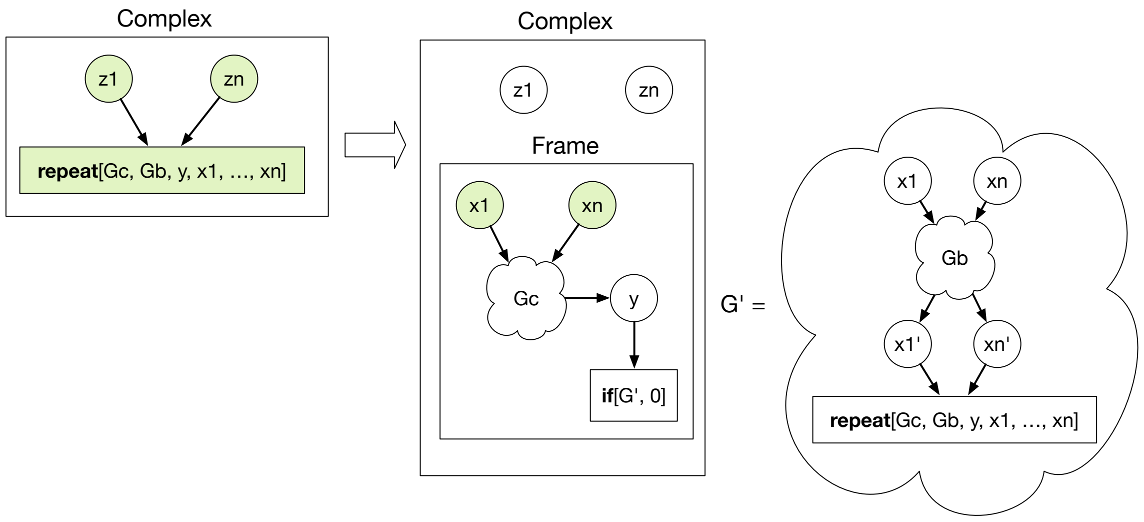
The new snapshot S' is defined as follows:

$$S' = S\{\text{memory} \mapsto M'\{a \mapsto C'\}\}$$

where

- $C' = \{f_{new}\} \cup \{f'\} \cup (C - \{f\})$
- $f_{new} = f\{\text{control} \mapsto G_C \cup \textbf{if}[G', \emptyset](y)\}$
- $M' = update(\{x_1 \mapsto d_1, \dots, x_n \mapsto d_n\}, f, M)$
- $G' = G_B \cup \{\textbf{repeat}[G_C, G_B, y, x_1, \dots, x_n](x'_1, \dots, x'_n)\}$
- $f' = f\{\text{control} \mapsto (G - \{\textbf{repeat}[G_C, G_B, y, x_1, \dots, x_n](z_1, \dots, z_n)\})\}$

The Figure below shows an example of this transition, with the **repeat** instruction creating a new frame that computes and tests the condition, which if **true**, then executes the body of the loop and possibly another iteration, represented by another **repeat** instruction.


Complex
z1
zn
repeat[Gc, Gb, y, x1, ..., xn]
Complex
z1
zn
Frame
x1
xn
Gc
y
if[G', 0]
G' =
x1
xn
Gb
x1'
xn'
repeat[Gc, Gb, y, x1, ..., xn]

Call to a Subroutine

A **call** operation takes the following form.

$$\textbf{call}[sname](z_1, \dots, z_n)$$

The effect of this operation is to create a new frame that will be placed within the complex where the operation originates. The frame will execute in parallel with other frames of the same complex and with the ones of the other complexes and it may use local variables. Notice that the category of subroutine names ($sname$) is assumed to be disjoint with those ones of method names ($mname$) and procedure names ($pname$).

This transition rule can fire in snapshot S if the following conditions are met:

- $M = S(\text{memory}),\ M(a) = C^{a_p}, f \in C$
- $\textbf{call}[sname](z_1, \dots, z_n) \in f(\text{control})$
- $M(a_p)(\text{methods})(sname) = \{\text{params} \mapsto PS, \text{body} \mapsto G\}$
- $PS = \{x_1 \mapsto T_1, \dots, x_n \mapsto T_n\}$
- $lookup(z_1, f, M) = d_1$ and ..., and $lookup(z_n, f, M) = d_n$

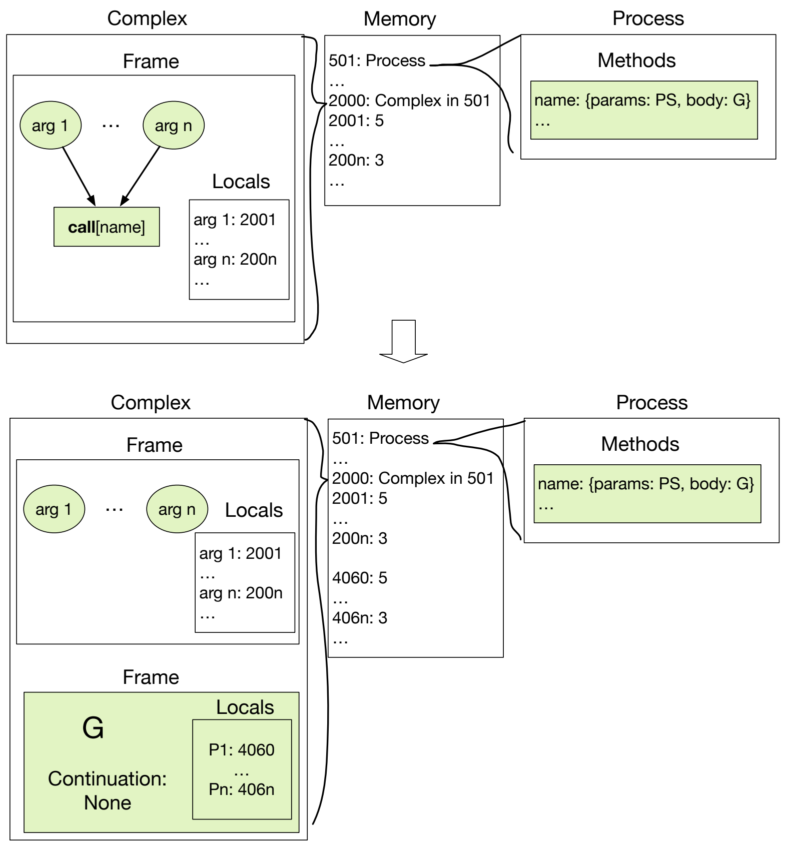
The new snapshot S' is defined as follows:

$$S' = S\{\text{memory} \mapsto M'\{a \mapsto C'^{a_p}\}\}$$

where

- $f_{new} = f\{\text{control} \mapsto G, \text{continuation} \mapsto \text{None}\}$
- $M' = update(\{x_1 \mapsto d_1, \dots, x_n \mapsto d_n\}, f, M)$
- $C' = \{f_{new}\} \cup \{f'\} \cup (C - \{f\})$
- $f' = f\{\text{control} \mapsto f(\text{control}) - \{\textbf{call}[sname](z_1, \dots, z_n)\}\}$

The Figure below shows an example call to a subroutine. A new frame is added to the complex to execute the dataflow graph G associated with the subroutine.



Array Creation

The $z \leftarrow \textbf{array}[T, m](x, y)$ operation initializes an array of elements of type T. The size of the array is contained in x and each element can be accessed with multiplicity m. A new address is assigned to the array, i.e. to the first element of the array. The following elements are placed at addresses that are contiguous in the memory starting from the address of the first element. The array is placed in the same locality as the "hint" argument y, which is optional. If there is no hint argument, the locality is the same as the compute complex that executes the operation. The address of the array is returned into z.

This transition rule can fire in snapshot S if the following conditions are met:

- $M = S(\text{memory}), fetch\_instruction(M) = (instr, f^{a_p}, M')$
- $instr = z \leftarrow \textbf{array}[T, m](x, y)$
- $lookup(x, f, M') = d_x$ and $lookup(y, f, M') = a_y$
- $M'\left(a_y\right) = v_\ell$
- $access\_control(a_p, a_y)$

The new snapshot S' is defined as follows:

$$S^{'} = S\{\text{memory} \mapsto M^{''}\left\{a \mapsto C^{'}, a^{'} \mapsto \blacksquare_{\ell}^{m,a_p}, a^{'} + 1 \mapsto \blacksquare_{\ell}^{m,a_p}, \dots, a^{'} + d_x \mapsto \blacksquare_{\ell}^{m,a_p}\right\}\}$$

where

- $distinct(\{a', a' + 1, \dots, a' + d_x\}, \text{dom}(M))$
- $M'' = update(\{z \mapsto a'\}, f, M')$

Array Access: read operation

The $z_1, z_2 \leftarrow \textbf{readAt}(x, y)$ operation accesses the array at address y in order to read its element at the index in x. This value is returned into z_1. As for the load operation, we facilitate our typing discipline by returning the address being accessed, in particular it is returned into z_2 (see the Discipline section).

This transition rule can fire in snapshot S if the following conditions are met:

- $M = S(\text{memory}), fetch\_instruction(M) = (instr, f^{a_p}, M')$
- $instr = z_1, z_2 \leftarrow \textbf{readAt}(x, y)$
- $lookup(x, f, M') = d_x$ and $lookup(y, f, M') = a'$
- $M'(a' + d_x) = d$
- $access\_control(a_p, a')$

The new snapshot S' is defined as follows:

$$S' = S\{\text{memory} \mapsto update(\{z_1 \mapsto d, z_2 \mapsto a' + d_x\}, f, M')\}$$

Array Access: write operation

The $z' \leftarrow \textbf{writeAt}(x, y, z)$ operation accesses the array at address y in order to set its element at the index in x with the value in z. In order to facilitate our typing discipline, the address being accessed is returned into z' (see the Discipline section). If the multiplicity of the memory cell is 1 (write-one), then the memory cell must be empty ($\blacksquare$) to perform the store.

This transition rule can fire in some snapshot S if the following conditions are met:

- $M = S(\text{memory}), fetch\_instruction(M) = (instr, f^{a_p}, M')$
- $instr = z' \leftarrow \textbf{writeAt}(x, y, z)$
- $lookup(x, f, M') = d_x$, $lookup(y, f, M') = a'$ and $lookup(z, f, M') = d$
- $M'(a' + d_x) = w_\ell^m$ and $m = 1$ implies $w = \blacksquare$
- $access\_control(a_p, a')$

The new snapshot S' is defined as follows:

$$S' = S\{\text{memory} \mapsto M''\{(a' + d_x) \mapsto d_\ell^m\}\}$$

where

- $M'' = update(\{z' \mapsto a' + d_x\}, f, M')$

Migration

The runtime system implementing ParalleX may move entities from one locality to another for the purposes of reducing communication and load balancing. Thus, we include the following transition. In snapshot S, suppose there is an object in locality ℓ.

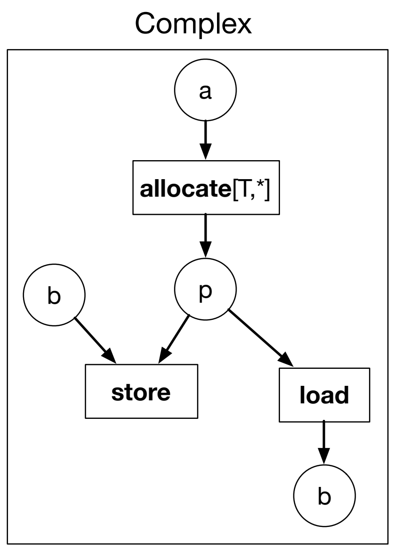
$$M(a) = v_{\ell}^{m,a_p} \text{ where } M = S(\text{memory})$$

Then the object can be relocated to another locality ℓ'.

$$S' = S\{\text{memory} \mapsto M\{a \mapsto v_{\ell'}^{m,a_p}\}\}$$

6. Discipline

Data races arise from unsynchronized accesses to the same address where at least one of the accesses is a write. For example, the below dataflow graph contains a data race because the **store** and **load** instructions can execute in parallel on the same memory address.



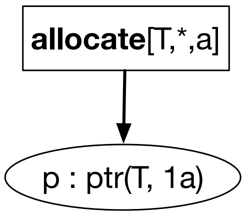
To prevent data races, ParalleX programs follow a static discipline called *fractional permissions* (J. T. Boyland, 2010; J. Boyland, 2003). This approach associates a numeric value between 0 and 1 (inclusive), a fraction, with every pointer. The fractions for the pointers to the same address are required to sum to exactly 1 at any point in time. In general, a pointer with a fraction of 1 is guaranteed to be the only pointer to its address and can therefore perform a write without concern for a data race. A pointer with a fraction of less than 1 is not allowed to write. A pointer with any fraction greater than zero can be used to read from its address because there cannot be another pointer that has a fraction of 1, that is, no other pointer can be used to write to the address.

The above description applies to pointers to mutable cells. The story for pointers to register cells is slightly different because registers have built-in synchronization.

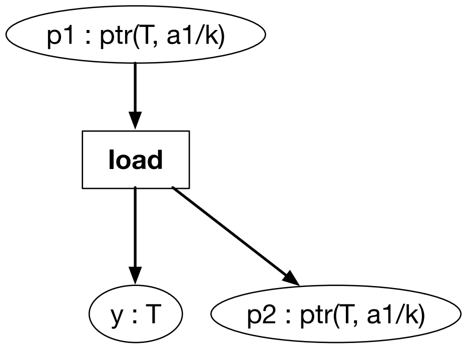
With registers, we still need to make sure that there are no concurrent writes to the same register, but we can allow reads to be concurrent with the single writer because the register causes reads to block until the write is complete. Writing to a register only requires a 2/3 or greater permission. As usual, reading requires any positive (non-zero) permission.

In the context of a ParalleX method specification, we define a set of rules to track how the fractional permissions flow through the dataflow graph. Each of the rules is expressed as a node of the dataflow graph with fractions on the inputs and outputs. Each fraction is associated with a unique identifier, a letter, which we call a *symbolic address*. There is a one-to-one correspondence between symbolic addresses and **allocate** operations, so we annotate the **allocate** node with the symbolic address. The symbolic address is representative of all the actual addresses that would be produced at runtime by that **allocate** operation. (For example, the **allocate** operation could appear inside a loop.) The fractions on the inputs represent lower bounds on what is allowed. The fractions on the outputs specify how the output fraction is computed from the input fractions.

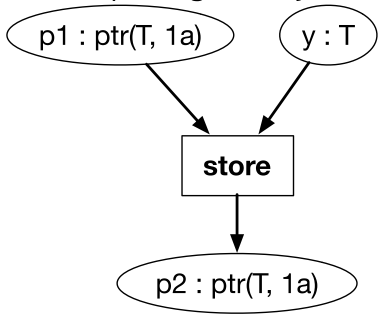
The first rule we consider is for the **allocate** operation, shown below. The **allocate** node has been annotated with a unique identifier, in this case '*b'*. The output *p* has a fraction of 1 that is associated with this symbolic address *b*. Thus, the output can be used for reading and writing.



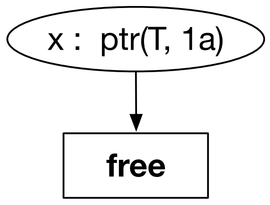
Next we consider the rule for the **load** operation. The input *p1* must be a pointer with non-zero permission (*1/n*) and the output *p2* is the same symbolic address with the same permission. The output *y* has the pointed-to type *T* from *p1.*



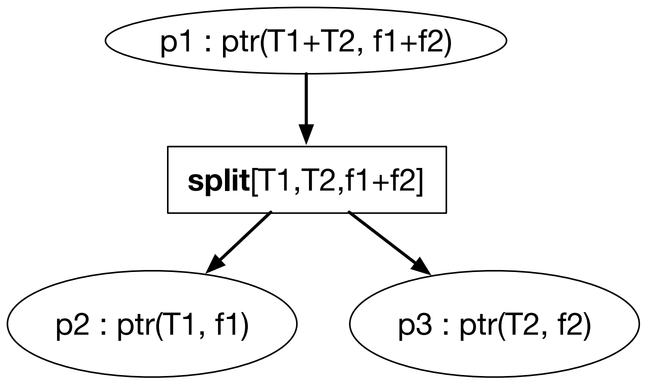
The **store** operation requires permission 1 from the input pointer *p1*. The type for input *x* must match the pointed-to type *T* for *p1*. The output *p2* has the same symbolic address as *p1* with the same permission 1. (If the pointer *p1* is to a register cell, then the permission must be 2/3 or greater.)



The **free** operation requires permission 1 from the input pointer *x*, as shown in the diagram below.



To prevent the unrestricted duplication of permissions, we depart from Boyland's approach and require that each *variable* has only a single out edge, that is, it is only used once. This is called a *affine* variable (Walker, 2005). However, to use the result of a computation more than once, a program can use a **split** instruction to duplicate the value and also make sure that the permissions are divided appropriately. We define a novel addition operator on types, used below in the type of p1, to control how permissions are divided when the program contains nested pointers.



To combine permissions, a program can use the **join** instruction, shown below.

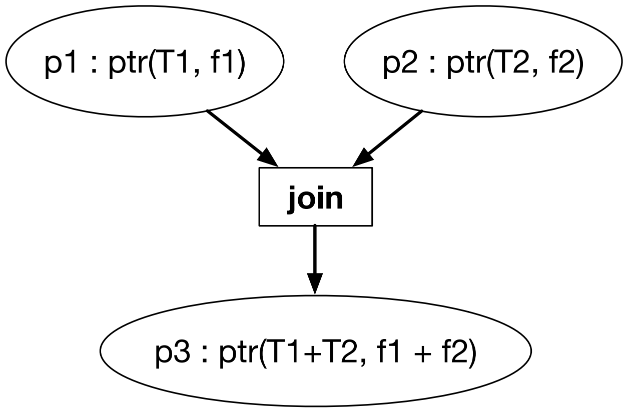


Figure **1** shows an example dataflow graph that satisfies the rules. In such a situation we say that the dataflow graph is *well-typed*, even though the rules talk about more than just types. In the example, we allocate a mutable memory cell, store the integer 7 into it, load from it twice (into variables *x* and *z*), add those numbers, and store the result back into the same memory cell. Thus, at the end of this dataflow graph, the memory cell contains 14. Even though this program contains several loads and stores, there are no data races. The store of 7 is guaranteed to occur before the two loads and the two loads are guaranteed to happen before the store.

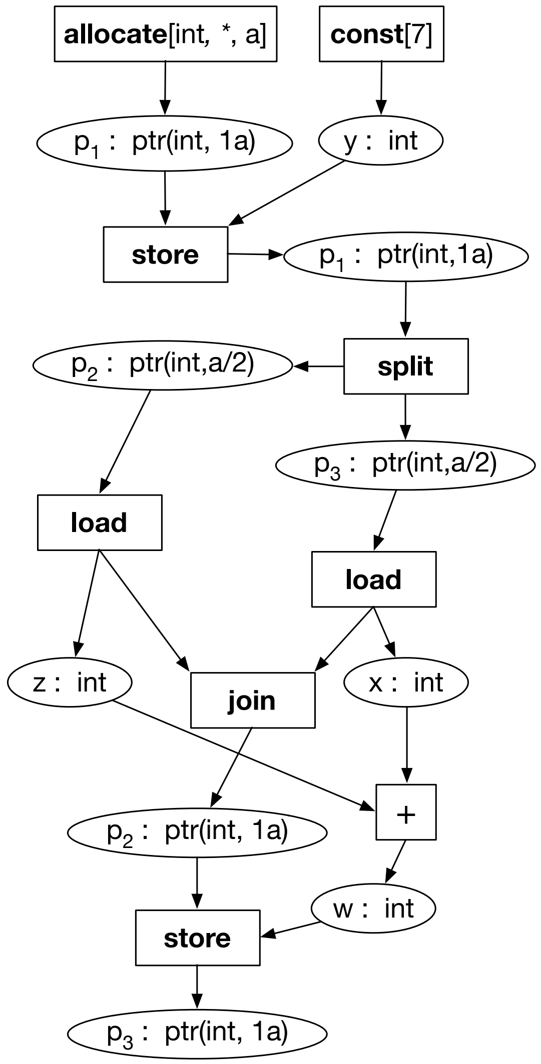


Figure 1. Example of a well-typed dataflow graph

7. Conclusion

This document formally defines the basic entities and actions of the ParalleX execution model. The definition is an operational semantics that specifies an abstract notion of a snapshot of an entire parallel system and it specifies how the system can evolve by transitioning from one snapshot to the next. The semantics is inherently nondeterministic in that ParalleX enables high degrees of parallelism, but much of the difficulty that stems from non-determinism, i.e. data races, is controlled by a discipline of static checking based on fractional permissions.

References

Arvind, Nikhil, R., & Pingali, K. (1987). I-Structures: Data structures for parallel computing. In J. Fasel & R. Keller (Eds.), *Graph Reduction* (Vol. 279, pp. 336–369). Springer Berlin Heidelberg. http://doi.org/10.1007/3-540-18420-1_65

Boyland, J. (2003). Checking Interference with Fractional Permissions. In *Proceedings of the 10th International Conference on Static Analysis* (pp. 55–72). Berlin, Heidelberg: Springer-Verlag. Retrieved from http://dl.acm.org/citation.cfm?id=1760267.1760273

Boyland, J. T. (2010). Semantics of Fractional Permissions with Nesting. *ACM Trans. Program. Lang. Syst.*, *32*(6), 22:1–22:33. http://doi.org/10.1145/1749608.1749611

Dennis, J. (1974). First version of a data flow procedure language. In B. Robinet (Ed.), *Programming Symposium* (Vol. 19, pp. 362–376). Springer Berlin Heidelberg. http://doi.org/10.1007/3-540-06859-7_145

Felleisen, M., & Hieb, R. (1992). The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, *103*(2), 235–271.

Friedman, D. P., & Wise, D. S. (1979). *Applicative Multiprogramming*.

Halstead Jr., R. H. (1985). MULTILISP: A Language for Concurrent Symbolic Computation. *ACM Trans. Program. Lang. Syst.*, *7*(4), 501–538. http://doi.org/10.1145/4472.4478

Kahn, G. (1987). Natural Semantics. In *Symposium on Theoretical Aspects of Computer Science* (pp. 22–39). Springer.

Landin, P. J. (1964). The Mechanical Evaluation of Expressions. *The Computer Journal*, *6*(4), 308–320.

Plotkin, G. D. (2004). A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, *60-61*, 17–139.

Walker, D. (2005). Substructural Type Systems. In B. C. Pierce (Ed.), *Advanced Topics in Types and Programming Languages*. MIT Press.