Semantic Foundations of Jade

Martin C. Rinard and Monica S. Lam
Computer Systems Laboratory
Stanford University, CA 94305

Abstract

Jade is a language designed to support coarse-grain parallelism on both shared and distributed address-space machines. Jade is data-oriented: a Jade programmer simply augments a sequential imperative program with declarations specifying how the program accesses data. A Jade implementation dynamically interprets the access specification to execute the program concurrently while enforcing the program's data dependence constraints, thus preserving the sequential semantics.

This paper describes the Jade constructs and defines both a serial and a parallel formal operational semantics for Jade. The paper proves that the two semantics are equivalent.

1 Introduction

Over the last decade, research in parallel architectures has led to many new parallel systems. These systems range from multiprocessors with shared address spaces, multi-computers with distributed address spaces, to networks of high-performance workstations. Furthermore, the development of high-speed interconnection networks makes it possible to connect the systems together, forming a tremendous computational resource. An effective way to use these machines is to partition a computation into coarse-grain tasks. The current language support for this computing environment is, however, rather primitive: programmers must explicitly manage the hardware resources using low level communication and synchronization primitives. This paper presents Jade, a new language designed to simplify the expression of coarse-grain parallelism.

This research was supported in part by DARPA contract N00039-91-C-0138.

Instead of using explicitly parallel constructs to create and synchronize concurrent tasks, Jade programmers use declarative constructs to specify how parts of sequential program access data. The Jade implementation dynamically interprets these access specifications to determine which operations can execute concurrently without violating the program's sequential semantics. This data-oriented approach simplifies the programming process by preserving the familiar sequential, imperative programming paradigm. Jade programmers need not struggle with phenomena such as data races, deadlock and nondeterministic program behavior.

Jade is a set of extensions to existing sequential languages. Programmers can therefore parallelize large existing applications simply by analyzing how the program uses data and augmenting the source code with Jade extensions. Because Jade hides the low-level coordination of parallel activity from the programmer, these applications are portable across different parallel architectures.

We introduced the basic concepts of the data-oriented approach to concurrency in a previous paper [6]. The previous version of Jade was designed for machines with shared address spaces. We have implemented Jade as an extension to C, C++ and FORTRAN on the Encore Multimax, the Silicon Graphics IRIS 4D/240S, and Stanford DASH multiprocessor [7]. We have found it possible to parallelize sequential programs with a reasonable programming effort. Implemented applications include a parallel sparse Cholesky factorization algorithm due to Rothberg and Gupta [11], the Perfect Club benchmark MDG [1], LocusRoute, a VLSI routing system due to Rose [10], a parallel *make* program, and a program simulating the flow of smog in the Los Angeles basin.

We have revised the definition of Jade so that it can now be implemented on machines with separate address spaces. The same Jade program could be executed, for example, on both a shared address space multiprocessor and a network of workstations. This revision also makes it possible for the Jade implementation to dynamically verify the correctness of the access specifications. If the program does not correctly declare how it will access data, the Jade implementation will signal an error. This verification guarantees that the parallel and serial executions of a Jade program compute the same result.

This paper presents the revised Jade language, and establishes the semantic foundations for Jade. We first informally present the Jade constructs, and explain how a programmer uses the Jade access declaration statements to specify how parts of the program access data. As we present the Jade constructs, we describe the concurrency patterns that the data usage information generates. We then formally present both a sequential and a parallel operational semantics for Jade. Because Jade is a declarative language, it is not immediately obvious how the Jade implementation generates the parallel execution. The parallel operational semantics therefore provides insight into how to actually implement Jade. Finally, we prove that the sequential and parallel semantics are equivalent.

2 Jade Programming Paradigm

A Jade programmer provides the *program* knowledge required for efficient parallelization; the implementation combines its *machine* knowledge with this information to map the computation efficiently onto the underlying hardware. Here are the Jade programmer's responsibilities:

- *Task Decomposition:* The programmer starts with a serial program and uses Jade constructs to identify the program's task decomposition.
- *Data Decomposition:* The programmer determines the granularity at which tasks will access the data, and allocates data at that granularity.
- *Access Specification:* The programmer provides a dynamically determined specification of the data each task accesses.

The Jade implementation performs the following activities:

- *Constraint Extraction:* The implementation uses the program's serial execution order and the tasks' access specifications to extract the dynamic inter-task dependence constraints that the parallel execution must obey.
- *Synchronized Parallel Execution:* The implementation maps the tasks efficiently onto the hardware while enforcing the extracted dependence constraints.
- *Data Distribution:* On machines with multiple address spaces, the implementation generates the messages required to move data between processors.

2.1 Jade Data Model

Each Jade program has a shared memory that all tasks can access; objects (dynamically or statically) allocated in this memory are called *shared* objects. Each task also has a private memory consisting of a stack for procedure parameters and local variables and a heap for dynamically allocated objects accessed only by that task. Objects allocated in private memory are called *private* objects.

The implementation enforces the restriction that no shared object can contain a reference to a private object. This, along with the restriction that no task be directly given a reference to another task's private object, ensures that no task can access any other task's private objects.

Each task has an *access specification* which specifies how the task will access shared objects. The programmer defines a task's access specification using *access declaration statements*. For example, the `rd` (read) statement declares that the task may read the given object, while the `wr` (write) statement declares that the task may write the given object.

Accesses *conflict* if, to preserve the serial semantics, they must execute in the underlying sequential execution order. Accesses to different objects do not conflict. Writes to the same object conflict, while reads do not conflict. A read and a write to the same object also conflict. The Jade implementation exploits concurrency by relaxing the sequential execution order between tasks that declare no conflicting accesses.

Since the parallelization is based on the access specification, it is important that the access specification be accurate. An undeclared access could introduce a data race and make a parallel execution of the program compute an erroneous result. The Jade implementation precludes this possibility by dynamically checking each task's accesses to shared objects. If a task attempts to perform an undeclared access, the implementation will generate a run-time error.

The implementation serializes tasks that declare conflicting accesses to an object even though the tasks may actually access disjoint regions of the object. The programmer must therefore allocate objects at a fine enough granularity to expose the desired amount of concurrency in the program.

2.2 Basic Concurrency

Jade programmers use the `withonly-do` construct to identify a task and to specify how that task will access data. Here is the general syntactic form of the construct:

```
withonly { access declaration } do
         (parameters for task body) {
  task body
}
```

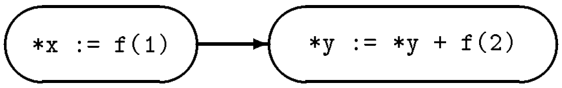
The `task body` section contains the serial code executed when the task runs. The `parameters` section contains a list of variables from the enclosing environment. When the task is created, the implementation copies the values of these variables into a new environment; the task will execute in this environment. To ensure that no task can reference another task's private objects, no variable in the parameters section can refer to a private object.

When a task is created, the Jade implementation executes the `access declaration` section to generate the task's access specification. This section is an arbitrary piece of code containing access declaration statements. Each such statement declares how the task will access a given shared object; the task's access specification is the union of all such declarations. This section may contain dynamically resolved variable references and control flow constructs such as conditionals, loops and function calls. The programmer may therefore use information available only at run time when generating a task's access specification.

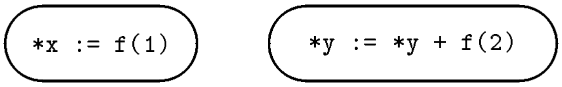
The Jade implementation uses the access specification information to execute the program concurrently while preserving the program's sequential semantics. We illustrate this concept by tracing the execution of the following Jade program.

```
x := sh(0);
y := sh(1);
if (g(0) > 0) {
  x := y;
};
withonly { wr(x); } do (x) {
  *x := f(1);
};
withonly { rd(y); wr(y); } do (y) {
  *y := *y + f(2);
}
```

This program first uses the `sh` construct to allocate two objects in the shared heap. `x` and `y` refer to these shared objects. The program then computes `g(0)`, making `x` and `y` refer to the same object if `g(0) > 0`. The program then executes the two `withonly-do` constructs. Each `withonly-do` construct creates a task and generates that task's access specification. If `x` and `y` refer to the same object, then the tasks' accesses may conflict because both specifications declare that the tasks will write the same object. In this case the implementation preserves the serial semantics by executing the first task before the second task. The program therefore generates the following sequential task graph:



If `x` and `y` refer to different objects, the two tasks' access specifications declare that the tasks will access disjoint sets of objects. Therefore, the two tasks' accesses do not conflict. The implementation can execute the tasks concurrently without violating the program's serial semantics. In this case the program generates the following parallel task graph:



Conceptually, the Jade implementation dynamically generates and executes a task graph. As the implementation creates tasks, it inserts each new task into the task graph. To preserve the serial semantics the implementation inserts precedence arcs between tasks whose accesses may conflict. Each such arc goes from the earlier task in the underlying sequential execution order to the later task. When a processor becomes idle it executes one of the initial tasks (a task with no incoming precedence arcs) in the current task graph. When the task completes it is removed from the task graph. By building the task graph incrementally at run time, the Jade implementation can detect and exploit dynamic, data-dependent concurrency available only as the program runs.

The programmer controls the amount of exploited concurrency by choosing the appropriate task granularity. Because the statements in a task execute sequentially, two pieces of code can execute concurrently only if they are in different tasks. The programmer must therefore make the task decomposition fine enough to expose the desired amount of concurrency.

2.3 Advanced Concurrency

In the model of parallel computation presented in section 2.2, a task's access specification is determined once and for all when the task is created. Two tasks may either execute concurrently (if none of their accesses conflict) or sequentially (if their accesses may conflict). Therefore, all synchronization takes place at task boundaries. The following example demonstrates how synchronizing only at task boundaries can waste concurrency.

```
x := sh(0);
y := sh(1);
withonly { wr(x); } do (x) {
  *x := f(1);
};
withonly { rd(y);rd(x);wr(x); }
  do (y,x) {
  s := g(*y);
  *x := h(*x, s);
};
withonly { wr(y); } do (y) {
  *y := f(2);
}
```

This program generates three tasks. The tasks must execute sequentially to preserve the serial semantics. However, the second task does not access `x` until it finishes the statement `s := g(*y)`. Therefore, the first task should be able to execute concurrently with the statement `s := g(*y)` from the second task. Similarly, the second task no longer accesses `y` after the statement `s := g(*y)` finishes. The statement `*x := h(*x, s)` from the second task should be able to execute concurrently with the third task. This example illustrates how information about when tasks access shared objects can expose concurrency.

To allow programmers to express when a task will access shared objects, Jade provides both a new construct, `with-cont`, and new access declaration statements `df_rd`, `df_wr`, `no_rd` and `no_wr`. The `with-cont` construct allows the programmer to update a task's access specification as the task executes. This construct, in combination with the new access declaration statements, allows the programmer to exploit the kind of inter-task concurrency described above.

Here is the general syntactic form of the `with-cont` construct:

```
with { access declaration } cont;
```

As in the `withonly-do` construct, the `access declaration` section is an arbitrary piece of code containing access declaration statements. These statements change the task's access specification so that it more precisely reflects how the rest (or continuation, as the `cont` keyword suggests) of the task will access shared objects.

The `with-cont` construct combines the current access specification with the declarations in the access declaration section to generate a new access specification. Unless the access declaration explicitly declares otherwise, the new access specification has the same declarations as the old access specification. The `withonly-do` construct, on the other hand, builds its access specification from scratch. Unless it explicitly declares an access, the access specification does not contain that declaration. The keywords in the constructs (`with` vs. `withonly`) reflect this difference in the treatment of undeclared accesses.

2.4 Deferred Accesses

The `df_rd` and `df_wr` statements declare a *deferred* access to the shared object. That is, they specify that the task may eventually read or write the object, but that it will not do so immediately. Before the task can access the object, it must execute a `with-cont` construct that uses the `rd` or `wr` access declaration statements to convert the deferred declaration to an *immediate* declaration. Therefore, a task that initially declares a deferred access to a shared object does not have the right to access that object. It does, however, have the right to convert the deferred declaration to an immediate declaration. This immediate declaration then gives the task the right to access the object.

Deferred declarations allow a task to defer its synchronization for a shared object until just before it actually accesses the object. The following modification to our example illustrates how deferred declarations can increase the amount of exploitable concurrency in a Jade program:

```
x := sh(0);
y := sh(1);
withonly { wr(x); } do (x) {
  x := f(1);
};
withonly { rd(y); df_rd(x); df_wr(x); }
  do (y,x) {
  s := g(*y);
  with { rd(x); wr(x); } cont;
  *x := h(*x, s);
};
withonly { wr(y); } do (y) {
  *y := f(2);
}
```

Because the second task declares a deferred read and a deferred write access on `x`, it cannot access `x` until it converts the deferred declarations to immediate declarations. The second task can therefore start to execute while the first task is still running. The `with-cont` statement in the second task converts the deferred declarations to immediate declarations. Because the immediate declarations give the second task the right to access `x`, it must wait until the first task completes before it can proceed. This example demonstrates how deferred declarations allow the

Jade implementation to execute an initial segment of one task concurrently with another task even though the second task may eventually carry out an access that conflicts with one of the first task's accesses.

2.5 Completed Accesses

Jade programmers use the `no_wr` and `no_rd` access declaration statements to explicitly remove a declaration from a task's access specification. These statements allow the programmer to indicate when a task has completed a specified access. The following example illustrates how the `no_wr` and `no_rd` statements can increase the amount of exploitable concurrency:

```
x := sh(0);
y := sh(1);
withonly { wr(x); } do (x) {
  *x := f(1);
};
withonly { rd(y); df_rd(x); df_wr(x); }
  do (y,x) {
  s := g(*y);
  with { no_rd(y); rd(x); wr(x); } cont;
  *x := h(*x, s);
};
withonly { wr(y); } do (y) {
  *y := f(2);
}
```

After the second task executes the `with-cont` statement, it no longer declares that it will read `y`. At this point the two tasks cannot perform conflicting accesses, so the rest of the second task can execute concurrently with the third task.

This example illustrates how programmers can use the `with-cont` construct and the `no_wr` and `no_rd` access declaration statements to eliminate conflicts between the enclosing task and tasks occurring later in the underlying sequential execution order. This conflict elimination may allow later tasks to execute as soon as the enclosing task executes the `with-cont` statement. In the absence of the `with-cont` statement the tasks would have had to wait until the enclosing task terminated.

2.6 Summary

Access specifications give the Jade implementation all the information it needs to correctly execute a program in parallel. Programmers generate a task's initial access specification when it is created, and can update the specification as the task runs. At any time, the current access specification must accurately reflect how the rest of the task and its future sub-tasks will access data. It is this *a priori* restriction - the guarantee that neither the task nor any of its sub-tasks will ever access certain shared objects - that allows the Jade implementation to exploit concurrency between tasks.

Each declaration *enables* a task and its sub-tasks to access a shared object in a certain way. For example, an immediate read declaration enables the task to read the data; it also enables the task's sub-tasks to declare a read access and then read the data. Conversely, if a task has not declared a read access on a given object, a sub-task cannot declare a read access and thus cannot read the object. A declaration therefore allows a sub-task to access certain data by enabling the sub-task to declare certain accesses. Table 1 summarizes the actions that read declarations enable. The table for writes is similar.

Declaration	Enabled Access	Enabled Declarations

`rd`	read	`rd`
		`df_rd`
		`no_rd`

`df_rd`	none	`rd`
		`df_rd`
		`no_rd`

`no_rd`	none	none

Table 1: Enabled Actions

When the implementation executes tasks, it must ensure that the parallel execution performs conflicting accesses in the same order as the serial execution. Therefore, a task cannot execute if any of its enabled accesses conflict with an immediate or deferred access declared by an earlier task.

3 Operational Semantics

Section 2 informally described the Jade language and the concurrent behavior of Jade programs. In this section we develop both a serial and a parallel operational semantics for Jade, and show that the parallel and serial semantics are equivalent.

Because Jade is a set of extensions to serial, imperative languages, we base our Jade semantics on a semantics for such a language. In this section we use the language Simple defined in Appendix A as our base language. We made Simple simple for purposes of exposition, but it is powerful enough to illustrate the basic concepts behind the Jade semantics. Although Simple only supports integers and references, our semantic treatment trivially generalizes to languages with more elaborate data structures. Simple also has no sophisticated flow of control constructs such as first class continuations or closures. Again, the semantic treatment presented in this section trivially generalizes to languages with such constructs. Because Jade only deals with a program's dynamic memory accesses, it does not matter what part of the program happens to generate these accesses.

We first define an operational semantics for Simple. This semantics consists of the definition of expression evaluation and the definition of a transition relation on Simple program states. This transition relation is in effect an interpreter for Simple. We use the transition relation for Simple to define a serial operational semantics for Jade programs. This semantics executes Jade programs in the standard serial execution order and dynamically checks the correspondence between each task's declared and actual accesses to shared objects.

We also use the transition relation for Simple to define the parallel operational semantics for Jade. The parallel and serial semantics use the same mechanism to check that tasks perform no undeclared accesses to shared objects. The parallel semantics, however, runs tasks concurrently if they can perform no conflicting accesses. The parallel semantics maintains a set of active tasks and a set of suspended tasks. Active tasks can execute concurrently; suspended tasks wait for active tasks to complete conflicting accesses. This semantics uses the standard interleaving approach to modelling concurrency in that it models the parallel execution of tasks by interleaving their atomic transitions.

The parallel semantics synchronizes tasks by maintaining, for each shared object, a queue of the declarations of tasks that may access that object. When all of a task's immediate declarations reach the front of their queues, the task is activated and can run. When a task terminates, it removes all of its declarations from the queues, potentially activating other tasks.

When a task is created, the implementation inserts each of its declarations into the appropriate queue just before its parent's declarations. When a task creates several sub-tasks, the sub-tasks' declarations will appear in the queue in their task creation order. This task creation order is also the standard serial execution order. Because a sub-task's declarations appear before its parent's declarations, the sub-task takes precedence over its parent. Therefore, tasks with conflicting declarations will get activated and execute in the standard serial execution order.

The main theoretical result of this paper is a theorem which establishes the correspondence between the serial semantics and the parallel semantics. The theorem states that a parallel execution of a Jade program will successfully halt if and only if the serial program successfully halts. Also, all such parallel and serial executions generate the same result.

The Appendix contains the complete set of axioms that define the operational semantics. In the paper we illustrate how the operational semantics work by reproducing representative axioms from the Appendix.

3.1 Access Specifications

In this section we formally define the access specifications that the semantics uses to check the correspondence between a task's declared and actual accesses. Each access specification is a set s of *declarations*. A declaration is a tuple of the form $\langle \mathsf{di} \in \{\mathtt{df}, \mathtt{im}, \mathtt{no}\}, \mathsf{rw} \in \{\mathtt{rd}, \mathtt{wr}\}, \mathsf{l}\rangle$. The first field of the tuple determines whether the tuple represents a deferred ($\mathtt{df}$), an immediate ($\mathtt{im}$) or no ($\mathtt{no}$) access declaration. The second field of the tuple determines whether the tuple represents a read ($\mathtt{rd}$) or a write ($\mathtt{wr}$) declaration, while the third field identifies the shared object to which the declaration refers. We say that a task with access specification s declares a deferred write access on a shared object l if $\langle \mathtt{df}, \mathtt{wr}, \mathsf{l}\rangle \in \mathsf{s}$, an immediate write access if $\langle \mathtt{im}, \mathtt{wr}, \mathsf{l}\rangle \in \mathsf{s}$, etc.

A task with access specification s can read (or write) a shared object l only if $\langle \mathtt{im}, \mathtt{rd}, \mathsf{l}\rangle \in \mathsf{s}$ (or $\langle \mathtt{im}, \mathtt{wr}, \mathsf{l}\rangle \in \mathsf{s}$). A task can declare an access (or no access) on an object in a `withonly-do` or `with-cont` construct only if s declares either an immediate or a deferred access on that object. We formalize the second concept with the following definition:

Definition 1
$\mathsf{s} \vdash \langle \mathsf{di}, \mathsf{rw}, \mathsf{l}\rangle$ *iff* $\langle \mathtt{df}, \mathsf{rw}, \mathsf{l}\rangle \in \mathsf{s}$ *or* $\langle \mathtt{im}, \mathsf{rw}, \mathsf{l}\rangle \in \mathsf{s}$.

The Jade implementation verifies that a task's access specification is accurate by dynamically checking every access to a shared object that the task declares or performs. If the task attempts to perform or declare an access that its access specification does not enable, the implementation detects the violation at run time and signals an error.

3.2 Expression Evaluation

In this section we define how Jade programs evaluate expressions. We assume an infinite set of shared memory locations ShLoc and an infinite set of private memory locations PrLoc. Each location holds a shared or private object. In Simple, shared locations can hold integers and references to shared objects. Private locations can hold integers and references to either shared or private objects. A memory is a partial function from a finite number of active memory locations to objects:

$$\begin{array}{rcl} \mathsf{l} & \in & \mathsf{Loc} = \mathsf{ShLoc} \cup \mathsf{PrLoc} \\ \mathsf{v} & \in & \mathsf{ShObj} = \mathsf{ShLoc} \cup \mathcal{Z} \\ \mathsf{v} & \in & \mathsf{PrObj} = \mathsf{ShObj} \cup \mathsf{PrLoc} \\ \mathsf{m} & \in & \mathsf{ShMem} = \mathsf{ShLoc} \stackrel{\text{fin}}{\to} \mathsf{ShObj} \\ \mathsf{n} & \in & \mathsf{PrMem} = \mathsf{PrLoc} \stackrel{\text{fin}}{\to} \mathsf{PrObj} \end{array}$$

Programmers use identifiers ($id \in Id$) to refer to variables. An environment e ($\in \mathsf{Env} = Id \stackrel{\text{fin}}{\to} \mathsf{PrObj}$) gives the correspondence between variables and values.

Expressions are evaluated in a context containing an environment, a shared and a private memory, and an access specification s. As described in section 3.1, s determines which shared objects a task has the right to read and/or write. The notation exp **in** $\langle \mathsf{e}, \mathsf{m}, \mathsf{n}, \mathsf{s}\rangle = \mathsf{v}$ should be read as: "expression exp in context $\langle \mathsf{e}, \mathsf{m}, \mathsf{n}, \mathsf{s}\rangle$ evaluates to v".

When a task evaluates an expression, it may read a shared object. At each such read, the semantics checks that the task declares an immediate read access on that object. This check takes the form of a precondition on the axiom that evaluates reads to shared objects. This axiom (reproduced below from Appendix B) requires that if a task reads a shared object, it must declare the access:

$$\frac{exp \ \mathbf{in}\ \langle \mathsf{e}, \mathsf{m}, \mathsf{n}, \mathsf{s}\rangle = \mathsf{l} \in \mathrm{Dom}\ \mathsf{m}, \langle \mathtt{im}, \mathtt{rd}, \mathsf{l}\rangle \in \mathsf{s}}{*exp \ \mathbf{in}\ \langle \mathsf{e}, \mathsf{m}, \mathsf{n}, \mathsf{s}\rangle = \mathsf{m}(\mathsf{l})}$$

Appendix B contains the complete set of axioms that define expression evaluation.

3.3 Simple Semantics

We now define the operational semantics for Simple programs. This semantics takes the form of a transition relation $\to$. Intuitively, each transition starts with a sequence of statements and a statement evaluation context and executes the first statement in the sequence. The result is a new statement sequence and a new context. Statement evaluations take place in the same contexts as expression evaluations.

A Simple statement may write a shared object. In this case the operational semantics must check that the task declares an immediate write access on that object. We reproduce that axiom here to show how the semantics uses a precondition to perform the check:

$$\frac{\begin{array}{c} exp_1 \ \mathbf{in}\ \langle \mathsf{e}, \mathsf{m}, \mathsf{n}, \mathsf{s}\rangle = \mathsf{l} \in \mathrm{Dom}\ \mathsf{m}, \langle \mathtt{im}, \mathtt{wr}, \mathsf{l}\rangle \in \mathsf{s}, \\ exp_2 \ \mathbf{in}\ \langle \mathsf{e}, \mathsf{m}, \mathsf{n}, \mathsf{s}\rangle = \mathsf{v} \in \mathsf{ShObj} \end{array}}{\begin{array}{c} *exp_1 := exp_2 ; c \ \mathbf{in}\ \langle \mathsf{e}, \mathsf{m}, \mathsf{n}, \mathsf{s}\rangle \to \\ c \ \mathbf{in}\ \langle \mathsf{e}, \mathsf{m}[\mathsf{l} \mapsto \mathsf{v}], \mathsf{n}, \mathsf{s}\rangle \end{array}}$$

If the task that generates the write does not declare the access, the Jade implementation must generate an error. Formally, the task takes a transition to the special state `error`. The semantics uses the following axiom to generate this transition:

$$\frac{exp_1 \ \mathbf{in}\ \langle \mathsf{e}, \mathsf{m}, \mathsf{n}, \mathsf{s}\rangle = \mathsf{l} \in \mathrm{Dom}\ \mathsf{m}, \langle \mathtt{im}, \mathtt{wr}, \mathsf{l}\rangle \notin \mathsf{s}}{*exp_1 := exp_2 ; c \ \mathbf{in}\ \langle \mathsf{e}, \mathsf{m}, \mathsf{n}, \mathsf{s}\rangle \to \mathtt{error}}$$

Finally, we demonstrate that when a task allocates a shared object, it acquires a deferred read and a deferred write declaration on the new object. The following axiom executes a `sh` statement, which allocates a new object:

$$\frac{\begin{array}{c} \mathsf{l} \in \mathsf{ShLoc} \setminus \mathrm{Dom}\ \mathsf{m}, exp \ \mathbf{in}\ \langle \mathsf{e}, \mathsf{m}, \mathsf{n}, \mathsf{s}\rangle = \mathsf{v} \in \mathsf{ShObj}, \\ \mathsf{s}' = \mathsf{s} \cup \{\langle \mathtt{df}, \mathtt{rd}, \mathsf{l}\rangle, \langle \mathtt{df}, \mathtt{wr}, \mathsf{l}\rangle\} \end{array}}{\begin{array}{c} id := \mathtt{sh}(exp) ; c \ \mathbf{in}\ \langle \mathsf{e}, \mathsf{m}, \mathsf{n}, \mathsf{s}\rangle \to \\ c \ \mathbf{in}\ \langle \mathsf{e}[id \mapsto \mathsf{l}], \mathsf{m}[\mathsf{l} \mapsto \mathsf{v}], \mathsf{n}, \mathsf{s}'\rangle \end{array}}$$

This axiom arbitrarily chooses a new location for the allocated object. Therefore, different executions of the serial program may differ in their choice of which locations hold which objects. Such executions represent equivalent computations, but the actual program states are different. To capture this equivalence we define what it means for two contexts to be equivalent:

Definition 2 $\langle \mathsf{e}_1, \mathsf{m}_1, \mathsf{n}_1, \mathsf{s}_1\rangle \equiv^b \langle \mathsf{e}_2, \mathsf{m}_2, \mathsf{n}_2, \mathsf{s}_2\rangle$ *iff* $\mathrm{Dom}\ \mathsf{e}_1 = \mathrm{Dom}\ \mathsf{e}_2$*, and there exist bijections* $b_n : \mathrm{Dom}\ \mathsf{n}_1 \to \mathrm{Dom}\ \mathsf{n}_2$*,* $b_m : \mathrm{Dom}\ \mathsf{m}_1 \to \mathrm{Dom}\ \mathsf{m}_2$ *and* $b_{\mathsf{s}} : \mathsf{s}_1 \to \mathsf{s}_2$ *such that* $b = b_n \cup b_m \cup \mathcal{I}$ *(where* $\mathcal{I}$ *is the identity function on* $\mathcal{Z} \cup \{\mathtt{error}\}$*) and*

1. $\forall \mathsf{l} \in \mathrm{Dom}\ \mathsf{n}_1 . b(\mathsf{n}_1(\mathsf{l})) = \mathsf{n}_2(b(\mathsf{l}))$*,*
2. $\forall \mathsf{l} \in \mathrm{Dom}\ \mathsf{m}_1 . b(\mathsf{m}_1(\mathsf{l})) = \mathsf{m}_2(b(\mathsf{l}))$*,*
3. $\forall id \in \mathrm{Dom}\ \mathsf{e}_1 . b(\mathsf{e}_1(id)) = \mathsf{e}_2(id)$*,*
4. $\forall \langle \mathsf{di}, \mathsf{rw}, \mathsf{l}\rangle \in \mathsf{s}_1 . b_{\mathsf{s}}(\langle \mathsf{di}, \mathsf{rw}, \mathsf{l}\rangle) = \langle \mathsf{di}, \mathsf{rw}, b(\mathsf{l})\rangle$*.*

Lemma 1 *If* $\langle \mathsf{e}_1, \mathsf{m}_1, \mathsf{n}_1, \mathsf{s}_1\rangle \equiv^b \langle \mathsf{e}_2, \mathsf{m}_2, \mathsf{n}_2, \mathsf{s}_2\rangle$ *then* $\forall exp \in Exp$*.*
$b(exp \ \mathbf{in}\ \langle \mathsf{e}_1, \mathsf{m}_1, \mathsf{n}_1, \mathsf{s}_1\rangle) = exp \ \mathbf{in}\ \langle \mathsf{e}_2, \mathsf{m}_2, \mathsf{n}_2, \mathsf{s}_2\rangle$*.*

Appendix C contains the rest of the axioms that define the operational semantics for Simple.

3.4 Jade Statement Semantics

This section defines the transition relation $\to_j$ for Jade statements. This transition relation extends $\to$ to the access declaration sections of Jade constructs. These transitions take place in Jade contexts; a Jade context is a tuple $\langle \mathsf{e}, \mathsf{m}, \mathsf{n}, \mathsf{s}, \mathsf{r}\rangle$. These contexts are the same as statement evaluation contexts, except that they contain an additional set of declarations r. The axioms accumulate declarations from the access declaration sections of Jade constructs into this set. When the semantics has finished executing the access declaration section of a `withonly-do` construct, r becomes the access specification of the new task. For a `with-cont` construct, the semantics uses r to update the current task's access specification.

We first reproduce an axiom that demonstrates how access declaration sections can contain arbitrary code. The following axiom makes all of the Simple transitions valid in the access declaration section of a `with-cont` construct:

$$\frac{c_1 \textbf{ in } \langle \mathsf{e}, \mathsf{m}, \mathsf{n}, \mathsf{s}\rangle \rightarrow c_1' \textbf{ in } \langle \mathsf{e}', \mathsf{m}', \mathsf{n}', \mathsf{s}'\rangle}{\begin{array}{c}\texttt{with } \{c_1\}\texttt{cont};c_2 \textbf{ in } \langle \mathsf{e}, \mathsf{m}, \mathsf{n}, \mathsf{s}, \mathsf{r}\rangle \rightarrow_j \\ \texttt{with } \{c_1'\}\texttt{cont};c_2 \textbf{ in } \langle \mathsf{e}', \mathsf{m}', \mathsf{n}', \mathsf{s}', \mathsf{r}\rangle\end{array}}$$

We next reproduce an axiom dealing with the accumulation of declarations into the access specification set r. To legally declare that a new task will access a given shared object, the parent task's access specification must enable the declaration. The semantics enforces this constraint with a precondition on the axiom which constructs the new task's access specification:

$$\frac{\begin{array}{c} c = \texttt{withonly } \{\mathsf{di\_rw}(exp);c_1\}\texttt{do}(ids)\{c_2\};c_3, \\ exp \textbf{ in } \langle \mathsf{e}, \mathsf{m}, \mathsf{n}, \mathsf{s}\rangle = \mathsf{l} \in \text{Dom } \mathsf{m}, \\ \mathsf{r}' = \mathsf{r} \cup \{\langle \mathsf{di}, \mathsf{rw}, \mathsf{l}\rangle\}, \mathsf{s} \vdash \langle \mathsf{di}, \mathsf{rw}, \mathsf{l}\rangle \end{array}}{\begin{array}{c} c \textbf{ in } \langle \mathsf{e}, \mathsf{m}, \mathsf{n}, \mathsf{s}, \mathsf{r}\rangle \rightarrow_j \\ \texttt{withonly } \{c_1\}\texttt{do}(ids)\{c_2\};c_3 \textbf{ in } \langle \mathsf{e}, \mathsf{m}, \mathsf{n}, \mathsf{s}, \mathsf{r}'\rangle \end{array}}$$

Appendix D contains the rest of the axioms which define the $\rightarrow_j$ transition relation, and the definition of the equivalence relation $\equiv_j^b$ for Jade contexts.

3.5 Serial Jade Semantics

We now define the transition relation $\rightarrow_s$ for serial Jade program states. A serial Jade program state $\langle \mathsf{m}, \mathsf{ts}\rangle = \mathsf{st} \in \mathsf{SerState} = \mathsf{ShMem} \times (\mathsf{Task}*)$ is a pair consisting of a shared store and a stack of tasks. Each task is a tuple $\mathsf{t} = \langle c, \mathsf{e}, \mathsf{n}, \mathsf{s}, \mathsf{r}\rangle \in \mathsf{Task}$ containing the code c for the task.

The following axiom defines the execution of a `withonly-do` statement. The precondition first checks that none of the task parameters refers to a private object. The new task then becomes the first task in the sequence, with the parent task second. Therefore, the program's statements execute in the standard sequential, depth-first execution order:

Definition 3

$$\begin{array}{rl} \mathsf{s} \uparrow \mathsf{r} = & \{\langle \mathsf{di}, \mathsf{rw}, \mathsf{l}\rangle \in \mathsf{s} \mid \neg \exists \langle \mathsf{di}', \mathsf{rw}, \mathsf{l}\rangle \in \mathsf{r}\} \cup \\ & \{\langle \texttt{im}, \mathsf{rw}, \mathsf{l}\rangle \in \mathsf{r}\} \cup \{\langle \texttt{df}, \mathsf{rw}, \mathsf{l}\rangle \in \mathsf{r}\}. \end{array}$$

$$\frac{\begin{array}{c} c = \texttt{withonly } \{\epsilon\}\texttt{do}(id_1, \ldots, id_n)\{c_1\};c_2, \\ \forall i \leq n. id_i \in \text{Dom } \mathsf{e} \text{ and } \mathsf{e}(id_i) \notin \mathsf{PrLoc}, \\ \mathsf{e}' = [id_1 \mapsto \mathsf{e}(id_1)] \cdots [id_n \mapsto \mathsf{e}(id_n)], \end{array}}{\begin{array}{c} \langle \mathsf{m}, \langle c, \mathsf{e}, \mathsf{n}, \mathsf{s}, \mathsf{r}\rangle \circ \mathsf{ts}\rangle \rightarrow_s \\ \langle \mathsf{m}, \langle c_1, \mathsf{e}', \emptyset, \emptyset \uparrow \mathsf{r}, \emptyset\rangle \circ \langle c_2, \mathsf{e}, \mathsf{n}, \mathsf{s}, \emptyset\rangle \circ \mathsf{ts}\rangle \end{array}}$$

When a task completes, the computation continues with the rest of its parent task. The next axiom removes completed tasks from the top of the stack of tasks. The completed task's parent is the new first task, so the program's execution continues with the parent task:

$$\overline{\langle \mathsf{m}, \langle \epsilon, \mathsf{e}, \mathsf{n}, \mathsf{s}, \mathsf{r}\rangle \circ \mathsf{ts}\rangle \rightarrow_s \langle \mathsf{m}, \mathsf{ts}\rangle}$$

When a program successfully halts, it takes a transition to the integer that is the program's result:

$$\frac{c = \texttt{result}(exp), exp \textbf{ in } \langle \mathsf{e}, \mathsf{m}, \mathsf{n}, \mathsf{s}\rangle = \mathsf{v} \in \mathcal{Z}}{\langle \mathsf{m}, \langle c, \mathsf{e}, \mathsf{n}, \mathsf{s}, \mathsf{r}\rangle\rangle \rightarrow_s \mathsf{v}}$$

The rest of the axioms that define $\rightarrow_s$ appear in Appendix E. In particular, there is an axiom that takes the serial program state to `error` if there is an error in the execution of one of the tasks.

We now define the notion of equivalence for serial Jade program states, and state a theorem that allows us to treat $\rightarrow_s$ as a transition function between equivalence classes of serial Jade program states.

Definition 4 *$b \downarrow s$ is the restriction of b to s. That is,* Dom $b \downarrow s =$ Dom $b \cap s$ *and* $b \downarrow s(\mathsf{v}) = b(\mathsf{v})$.

Definition 5 $\langle \mathsf{m}, \mathsf{t}_1 \circ \cdots \circ \mathsf{t}_n\rangle \equiv_s \langle \mathsf{m}', \mathsf{t}_1' \circ \cdots \circ \mathsf{t}_n'\rangle$ *iff* b_{m} : Dom $\mathsf{m} \rightarrow$ Dom m' *is a bijection and* $\forall i \leq n.$ *if* $\mathsf{t}_i = \langle c, \mathsf{e}, \mathsf{n}, \mathsf{s}, \mathsf{r}\rangle, \mathsf{t}_i' = \langle c', \mathsf{e}', \mathsf{n}', \mathsf{s}', \mathsf{r}'\rangle$ *then* $c = c'$ *and* $\exists b.b \downarrow$ Dom $b_{\mathsf{m}} = b_{\mathsf{m}}, \langle \mathsf{e}, \mathsf{m}, \mathsf{n}, \mathsf{s}, \mathsf{r}\rangle \equiv_j^b \langle \mathsf{e}', \mathsf{m}, \mathsf{n}', \mathsf{s}', \mathsf{r}'\rangle$.

Theorem 1 *If* $\mathsf{st}_1 \equiv_s \mathsf{st}_2$ *then*

1. $\mathsf{st}_2 \equiv_s \mathsf{st}_1$,
2. $\mathsf{st}_1 \rightarrow_s \mathsf{st}_1' \Rightarrow \exists \mathsf{st}_2'.\mathsf{st}_2 \rightarrow_s \mathsf{st}_2'$,
3. $\mathsf{st}_1 \rightarrow_s \mathsf{st}_1',\ \mathsf{st}_2 \rightarrow_s \mathsf{st}_2' \Rightarrow \mathsf{st}_1' \equiv_s \mathsf{st}_2'$ *or* $\mathsf{st}_1' = \mathsf{st}_2'$.

We can now view $\rightarrow_s$ as a program execution function. The value of $\rightarrow_s$ is the unique equivalence class of program states obtained by executing the next step of the program. Our serial semantics is therefore deterministic.

We now define the notion of observation for the serial execution of Jade programs. The basic idea is that we start the program in a start state and run it until it can progress no further. If the program halts with an integer result, we observe the result. If the program halts in `error` or could only partially execute we observe `error`. If the program runs forever we observe $\perp$:

Definition 6 $\mathsf{sst}(c) = \langle \emptyset, \langle c, \emptyset, \emptyset, \emptyset, \emptyset\rangle\rangle$.

$$\mathsf{SObs}(c) = \begin{cases} \mathsf{v} & \textit{if } \mathsf{sst}(c) \rightarrow_s \cdots \rightarrow_s \mathsf{v} \in \mathcal{Z} \\ \texttt{error} & \textit{if } \mathsf{sst}(c) \rightarrow_s \cdots \rightarrow_s \texttt{error} \textit{ or} \\ & \quad \mathsf{sst}(c) \rightarrow_s \cdots \rightarrow_s \mathsf{st} \not\rightarrow_s, \mathsf{st} \in \mathsf{SerState} \\ \perp & \textit{if } \mathsf{sst}(c) \rightarrow_s \cdots \rightarrow_s \cdots \end{cases}$$

3.6 Parallel Jade Semantics

We now define the transition relation $\rightarrow_p$ for parallel Jade program states. A parallel program state $\mathsf{pt} = \langle \mathsf{m}, \mathsf{A}, \mathsf{S}, \sqsubset \rangle \in \mathsf{ParState}$ consists of a shared memory m, a set A of active tasks, a set S of suspended tasks and an ordering relation $\sqsubset$ on the declarations of the parallel program state.

3.6.1 Object Queues

The following definitions impose some consistency requirements on the structure of $\sqsubset$:

Definition 7 *Given a set T of tasks,* $\mathsf{decl}(T) = \biguplus_{\langle c,\mathsf{e},\mathsf{n},\mathsf{s},\mathsf{r}\rangle \in T} \mathsf{s}$.

Definition 8 *We say that $\sqsubset$ is consistent for $\mathsf{A} \cup \mathsf{S}$ iff*

1. $\sqsubset \subseteq \mathsf{decl}(\mathsf{A} \cup \mathsf{S}) \times \mathsf{decl}(\mathsf{A} \cup \mathsf{S})$,
2. $\mathsf{d}_1 \sqsubset \mathsf{d}_2$ *and* $\mathsf{d}_2 \sqsubset \mathsf{d}_3 \Rightarrow \mathsf{d}_1 \sqsubset \mathsf{d}_3$.
3. $\langle \mathsf{di}, \mathsf{rw}, \mathsf{l} \rangle \sqsubset \langle \mathsf{di}', \mathsf{rw}', \mathsf{l}' \rangle \Rightarrow \mathsf{l} = \mathsf{l}'$.
4. $\langle \mathsf{di}, \mathsf{rw}, \mathsf{l} \rangle \not\sqsubset \langle \mathsf{di}', \mathsf{rw}', \mathsf{l} \rangle$ *and* $\langle \mathsf{di}', \mathsf{rw}', \mathsf{l} \rangle \not\sqsubset \langle \mathsf{di}, \mathsf{rw}, \mathsf{l} \rangle$
   $\Leftrightarrow \exists \langle c, \mathsf{e}, \mathsf{n}, \mathsf{s}, \mathsf{r} \rangle \in \mathsf{A} \cup \mathsf{S}.$
   $\langle \mathsf{di}, \mathsf{rw}, \mathsf{l} \rangle \in \mathsf{s}$ *and* $\langle \mathsf{di}', \mathsf{rw}', \mathsf{l} \rangle \in \mathsf{s}$.

Given this definition, $\sqsubset$ represents a set of queues, one for each shared object. Each declaration $\langle \mathsf{di}, \mathsf{rw}, \mathsf{l} \rangle$ appears in the queue for l.

Declarations appear in a queue in their tasks' underlying sequential execution order. So, if task t_1 would execute before task t_2 if the program executed sequentially, then t_1's declarations appear before the declarations of t_2. The operational semantics uses these queues to determine when tasks can execute concurrently. As soon as all of a task's immediate declarations reach the front of their queues, that task can execute. Therefore, if the declarations of two tasks are simultaneously at the front of their respective queues, the two tasks' access specifications do not conflict and the tasks can execute concurrently. We formalize the notion of "front of a queue" with the following definitions. $\mathsf{f}(\langle \mathsf{di}, \mathsf{rw}, \mathsf{l} \rangle, \sqsubset)$ is true just when $\langle \mathsf{di}, \mathsf{rw}, \mathsf{l} \rangle$ is at the front of its queue.

Definition 9
$\mathsf{succ}(\mathsf{s}, \sqsubset) = \{\mathsf{d}' \mid \exists \mathsf{d} \in \mathsf{s}.\mathsf{d} \sqsubset \mathsf{d}'\}$.
$\mathsf{pred}(\mathsf{s}, \sqsubset) = \{\mathsf{d}' \mid \exists \mathsf{d} \in \mathsf{s}.\mathsf{d}' \sqsubset \mathsf{d}\}$.
$\mathsf{f}(\langle \mathsf{di}, \mathtt{wr}, \mathsf{l} \rangle, \sqsubset)$ *iff* $\mathsf{pred}(\{\langle \mathsf{di}, \mathtt{wr}, \mathsf{l} \rangle\}, \sqsubset) = \emptyset$.
$\mathsf{f}(\langle \mathsf{di}, \mathtt{rd}, \mathsf{l} \rangle, \sqsubset)$ *iff*
$\forall \langle \mathsf{di}', \mathsf{rw}', \mathsf{l}' \rangle \in \mathsf{pred}(\{\langle \mathsf{di}, \mathtt{rd}, \mathsf{l} \rangle\}, \sqsubset).\mathsf{rw}' = \mathtt{rd}$.

As the definition of f shows, a write declaration is at the front of its queue when there are no declarations before it in the queue; a read declaration is at the front of its queue when there are only other read declarations before it in the queue. This reflects the fact that several tasks can concurrently read a shared object because reads do not change the object's state. Writes, of course, must execute in the underlying sequential execution order.

The definition of f demonstrates that a deferred declaration can prevent an immediate declaration from being at the front of its queue. In this case the deferred declaration prevents the immediate declaration's task from executing. This reflects the fact that a deferred declaration represents a potential access. The immediate declaration's task cannot proceed until there is no possibility that an earlier task can perform an access that conflicts with any of its accesses.

We now discuss the definition of $\rightarrow_p$, the transition relation for parallel Jade program states. We model the parallel execution of a Jade program by interleaving the atomic transitions of that program's parallel tasks. $\rightarrow_p$ therefore arbitrarily picks one of the active tasks and executes the next step of that task's computation. The tasks themselves may change their specifications, create new tasks, or complete their execution. Each of these events changes the program state's set of specifications. The transition relation must therefore modify $\sqsubset$ to reflect these changes.

There is a function to perform each kind of modification to $\sqsubset$. When the semantics executes a `withonly-do` construct, it uses the ins function to insert the new task's declarations into the queues just before its parent's declarations. When the task completes, the semantics uses the rem function to remove its declarations from the queues. When the semantics executes a `with-cont` construct, it uses the upd function to perform the queue modifications that correspond to the changes in the task's access specification.

Definition 10
$\mathsf{s@l} = \{\langle \mathsf{di}, \mathsf{rw}, \mathsf{l} \rangle \in \mathsf{s}\}$.
$\mathsf{rpl}(\mathsf{s}, \mathsf{r}, \sqsubset) =$
$\bigcup_{\langle \mathsf{di}, \mathsf{rw}, \mathsf{l} \rangle \in \mathsf{r}} (\mathsf{pred}(\mathsf{s@l}, \sqsubset) \times \mathsf{r@l}) \cup (\mathsf{r@l} \times \mathsf{succ}(\mathsf{s@l}, \sqsubset))$.
$\mathsf{ins}(\mathsf{r}, \mathsf{s}, \sqsubset) = \sqsubset \cup \mathsf{rpl}(\mathsf{s}, \mathsf{r}, \sqsubset) \cup_{\langle \mathsf{di}, \mathsf{rw}, \mathsf{l} \rangle \in \mathsf{r}} \mathsf{r@l} \times \mathsf{s@l}$.
$\mathsf{rem}(\mathsf{s}, \sqsubset) = \sqsubset \setminus \{\langle \mathsf{d}, \mathsf{d}' \rangle \mid \mathsf{d} \in \mathsf{s}$ *or* $\mathsf{d}' \in \mathsf{s}\}$.
$\mathsf{upd}(\mathsf{s}, \mathsf{r}, \sqsubset) = \mathsf{rem}(\mathsf{s}, \sqsubset) \cup \mathsf{rpl}(\mathsf{s}, \mathsf{r}, \sqsubset)$.

3.6.2 Transition Relation

We now present the axioms that define $\rightarrow_p$. We first present the axiom that executes a `withonly-do` statement when it creates a new task. This axiom suspends both the new task and its parent. The parent task may be unable to run because its access specification may conflict with the new task's access specification. The new task may be unable to run because its access specification may conflict with those of previously created tasks.

$$\frac{\begin{array}{c} c = \texttt{withonly}\ \{\epsilon\}\mathsf{do}(id_1, \ldots, id_n)\{c_1\};c_2, \\ \mathsf{t} = \langle c, \mathsf{e}, \mathsf{n}, \mathsf{s}, \mathsf{r}\rangle \in \mathsf{A}, \\ \forall i \leq n.id_i \in \mathrm{Dom}\ \mathsf{e} \text{ and } \mathsf{e}(id_i) \notin \mathsf{PrLoc} \\ \mathsf{e}' = [id_1 \mapsto \mathsf{e}(id_1)] \cdots [id_n \mapsto \mathsf{e}(id_n)], \\ \mathsf{t}' = \langle c_1, \mathsf{e}', \emptyset, \emptyset \uparrow \mathsf{r}, \emptyset\rangle, \mathsf{t}'' = \langle c_2, \mathsf{e}, \mathsf{n}, \mathsf{s}, \emptyset\rangle \end{array}}{\begin{array}{c}\langle \mathsf{m}, \mathsf{A}, \mathsf{S}, \sqsubset\ \rangle \rightarrow_p \\ \langle \mathsf{m}, \mathsf{A} \setminus \{\mathsf{t}\}, \mathsf{S} \uplus \{\mathsf{t}', \mathsf{t}''\}, \mathsf{ins}(\emptyset \uparrow \mathsf{r}, \mathsf{s}, \sqsubset)\rangle\end{array}}$$

When all of a suspended task's immediate declarations reach the front of their respective queues, the semantics must transfer the task to the set of active tasks so that it can execute. The following axiom activates such suspended tasks:

$$\frac{\begin{array}{c}\mathsf{t} = \langle c, \mathsf{e}, \mathsf{n}, \mathsf{s}, \mathsf{r}\rangle \in \mathsf{S}, \\ \forall \langle \texttt{im}, \mathsf{rw}, \mathsf{l}\rangle \in \mathsf{s.f}(\langle \texttt{im}, \mathsf{rw}, \mathsf{l}\rangle, \sqsubset)\end{array}}{\langle \mathsf{m}, \mathsf{A}, \mathsf{S}, \sqsubset\ \rangle \rightarrow_p \langle \mathsf{m}, \mathsf{A} \uplus \{\mathsf{t}\}, \mathsf{S} \setminus \{\mathsf{t}\}, \sqsubset\ \rangle}$$

When a task completes, it must remove its declarations from the queues. The semantics can then activate tasks whose accesses conflicted with the completed task's accesses.

$$\frac{\mathsf{t} = \langle \epsilon, \mathsf{e}, \mathsf{n}, \mathsf{s}, \mathsf{r}\rangle \in \mathsf{A},}{\langle \mathsf{m}, \mathsf{A}, \mathsf{S}, \sqsubset\ \rangle \rightarrow_p \langle \mathsf{m}, \mathsf{A} \setminus \{\mathsf{t}\}, \mathsf{S}, \mathsf{rem}(\mathsf{s}, \sqsubset\ )\rangle}$$

The rest of the axioms that define $\rightarrow_p$ are in Appendix F. In particular, there is an axiom that takes a program to `error` if the program violates some of the execution constraints, and an axiom that computes the program's result.

We next define how to observe the parallel execution of a Jade program. If the program successfully halts, we observe the result. If the program has an error, or gets into a state from which it cannot progress, or has one of its active tasks get into a state from which it cannot progress, we observe `error`. If the program runs forever, we observe $\perp$. The parallel observation function PObs observes every parallel execution and takes the union of the resulting observations. In this definition, PObs makes no fairness assumptions about the parallel execution.

Definition 11 $\mathsf{pst}(c) = \langle \emptyset, \{\langle c, \emptyset, \emptyset, \emptyset, \emptyset\rangle\}, \emptyset, \emptyset\rangle$.
$\mathsf{hung}(\langle \mathsf{m}, \mathsf{A}, \mathsf{S}, \sqsubset\ \rangle)$ *iff*
$\langle \mathsf{m}, \mathsf{A}, \mathsf{S}, \sqsubset\ \rangle \not\rightarrow_p$ *or* $\exists \mathsf{t} \in \mathsf{A}.\langle \mathsf{m}, \{\mathsf{t}\}, \emptyset, \sqsubset\ \rangle \not\rightarrow_p$.
$\mathsf{PObs}(c) =$

$$\begin{array}{lll} & \{\mathsf{v} \mid \mathsf{pst}(c) \rightarrow_p \cdots \rightarrow_p \mathsf{v} \in \mathcal{Z}\} & \\ \cup & \{\texttt{error}\} & \textit{if } \mathsf{pst}(c) \rightarrow_p \cdots \rightarrow_p \texttt{error} \textit{ or} \\ & & \quad \mathsf{pst}(c) \rightarrow_p \cdots \rightarrow_p \mathsf{pt}, \mathsf{hung}(\mathsf{pt}) \\ \cup & \{\perp\} & \textit{if } \mathsf{pst}(c) \rightarrow_p \cdots \rightarrow_p \cdots \end{array}$$

We next define a notion of consistency for parallel program states, and prove that $\rightarrow_p$ preserves consistency. We use lemma 2 extensively in the proof of correspondence between the serial and parallel semantics.

Definition 12 *A parallel program state* $\langle \mathsf{m}, \mathsf{A}, \mathsf{S}, \sqsubset\ \rangle$ *is consistent iff* $\sqsubset$ *is consistent for* $\mathsf{A} \cup \mathsf{S}$ *and* $\forall \langle c, \mathsf{e}, \mathsf{n}, \mathsf{s}, \mathsf{r}\rangle \in \mathsf{A} \cup \mathsf{S}$

1. $\forall \langle \mathsf{di}, \mathsf{rw}, \mathsf{l}\rangle \in \mathsf{r.l} \in \mathrm{Dom}\ \mathsf{m}, \mathsf{s} \vdash \langle \mathsf{di}, \mathsf{rw}, \mathsf{l}\rangle$
2. $\forall \langle \mathsf{di}, \mathsf{rw}, \mathsf{l}\rangle \in \mathsf{s.l} \in \mathrm{Dom}\ \mathsf{m}$*,* $\mathsf{di} \in \{\texttt{df}, \texttt{im}\}$*,*
3. $\langle c, \mathsf{e}, \mathsf{n}, \mathsf{s}, \mathsf{r}\rangle \in \mathsf{A} \Rightarrow$
   $\forall \langle \texttt{im}, \mathsf{rw}, \mathsf{l}\rangle \in \mathsf{s.f}(\langle \texttt{im}, \mathsf{rw}, \mathsf{l}\rangle, \sqsubset)$.

Lemma 2 *If* $\mathsf{pt} \in \mathsf{ParState}$ *is consistent,* $\mathsf{pt} \rightarrow_p \mathsf{pt}'$ *and* $\mathsf{pt}' \in \mathsf{ParState}$ *then* pt' *is consistent.*

Proof Sketch: The key aspect of the proof is to show that $\rightarrow_p$ preserves property 3 of definition 12. To show this, we must show that the legal queue updates and insertions do not cause the program state to violate this property.

We first consider the queue insertions caused by a parent task spawning a new task. For each queue, the new task's declarations appear just before the parent task's declarations. Both the parent task and the new task are suspended in the new state. All active tasks other than the parent task remain active in the new state.

We now show that any immediate declaration of an active task in the new state remains at the front of its queue. If there is no declaration from the parent task in the active task's declaration's queue, then by property 1 of definition 12 there is no declaration from the new task in that queue. Otherwise, the active task's declaration must have appeared either before or after the parent's declarations in the old state. If the declaration appeared before those of the parent task, then it will appear before those of the new task in the new state. If the declaration appeared after those of the parent task, then all of the declarations in question must be read declarations. By property 1 of definition 12 the new task inserted no write declarations in the queue, so the active task's read declaration is still at the front of the queue.

We next consider an update to the queue due to the execution of a `with-cont` statement. In the new state, the updating task's declarations appear in the same place in the queues as the task's declarations from the old state. We can use a case analysis similar to that for the insertion case to determine that the transition preserves property 3 of definition 12.

3.7 Semantic Correspondence

In this section we present the proof of correspondence between the parallel and serial semantics. We first set up an equivalence between serial program states and parallel program states. We use this definition to show that the serial execution of a Jade program is also one of the legal parallel executions. This is the first step towards proving the correspondence between the parallel and serial semantics.

Definition 13 $\langle \mathsf{m}, \mathsf{ts}\rangle \equiv_{sp} \langle \mathsf{m}, \mathsf{A}, \mathsf{S}, \sqsubset\rangle$ *iff there exists an ordering* $\langle c'_1, \mathsf{e}'_1, \mathsf{n}'_1, \mathsf{s}'_1, \mathsf{r}'_1\rangle \circ \cdots \circ \langle c'_m, \mathsf{e}'_m, \mathsf{n}'_m, \mathsf{s}'_m, \mathsf{r}'_m\rangle$ *on* $\mathsf{A} \cup \mathsf{S}$ *such that if* $\mathsf{ts} = \langle c_1, \mathsf{e}_1, \mathsf{n}_1, \mathsf{s}_1, \mathsf{r}_1\rangle \circ \cdots \circ \langle c_n, \mathsf{e}_n, \mathsf{n}_n, \mathsf{s}_n, \mathsf{r}_n\rangle$ *then* $m = n$ *and* $\forall i \leq n$

1. $c_i = c'_i, \mathsf{e}_i = \mathsf{e}'_i, \mathsf{n}_i = \mathsf{n}'_i, \mathsf{s}_i = \mathsf{s}'_i, \mathsf{r}_i = \mathsf{r}'_i$,
2. $\forall \langle \mathsf{di}, \mathsf{rw}, \mathsf{l}\rangle \in \mathsf{s}_i . \mathsf{f}(\langle \mathsf{di}, \mathsf{rw}, \mathsf{l}\rangle, \mathsf{rem}(\biguplus_{j<i} \mathsf{s}_j, \sqsubset))$.

Lemma 3 *If* $\mathsf{st} \equiv_{sp} \mathsf{pt}$, pt *is consistent and* $\mathsf{st} \rightarrow_s \mathsf{st}'$ *then* $\exists \mathsf{pt}' . \mathsf{pt} \rightarrow_p \cdots \rightarrow_p \mathsf{pt}'$ *and either* $\mathsf{st}' \equiv_{sp} \mathsf{pt}'$ *or* $\mathsf{st}' = \mathsf{pt}'$.

Proof Sketch: We perform a case analysis of all the transitions that a serial program state can take. We then identify a corresponding transition that an equivalent parallel program state can take, and show that the new serial and parallel program states are equivalent.

Theorem 2 $\mathsf{SObs}(c) \in \mathsf{PObs}(c)$.

Proof Sketch: A simple induction using lemmas 2 and 3.

We next define the notion of equivalence for parallel program states. This definition is again intended to capture precisely how two program states that differ only in their choice of allocated locations are equivalent.

Definition 14 $\langle \mathsf{m}_1, \mathsf{A}_1, \mathsf{S}_1, \sqsubset_1\rangle \equiv_p \langle \mathsf{m}_2, \mathsf{A}_2, \mathsf{S}_2, \sqsubset_2\rangle$ *iff there exist bijections* $b_a : \mathsf{A}_1 \rightarrow \mathsf{A}_2, b_s : \mathsf{S}_1 \rightarrow \mathsf{S}_2$, $b_m : \mathrm{Dom}\ \mathsf{m}_1 \rightarrow \mathrm{Dom}\ \mathsf{m}_2$ *and* $b_d : \mathsf{decl}(\mathsf{A}_1 \cup \mathsf{S}_1) \rightarrow \mathsf{decl}(\mathsf{A}_2 \cup \mathsf{S}_2)$ *such that if* $b_{as} = b_a \cup b_s$ *then*

$$\mathsf{d} \sqsubset_1 \mathsf{d}' \Leftrightarrow b_d(\mathsf{d}) \sqsubset_2 b_d(\mathsf{d}'),$$

and for all $\mathsf{t} = \langle c_1, \mathsf{e}_1, \mathsf{n}_1, \mathsf{s}_1, \mathsf{r}_1\rangle \in \mathsf{A}_1 \cup \mathsf{S}_1$, *if* $b_{as}(\mathsf{t}) = \langle c_2, \mathsf{e}_2, \mathsf{n}_2, \mathsf{s}_2, \mathsf{r}_2\rangle$ *then*

1. $c_1 = c_2$,
2. $\exists b.\ \langle \mathsf{e}_1, \mathsf{m}_1, \mathsf{n}_1, \mathsf{s}_1, \mathsf{r}_1\rangle \equiv^b_j \langle \mathsf{e}_2, \mathsf{m}_2, \mathsf{n}_2, \mathsf{s}_2, \mathsf{r}_2\rangle$ *and* $b \downarrow \mathrm{Dom}\ b_m = b_m$,
3. $\forall \langle \mathsf{di}, \mathsf{rw}, \mathsf{l}\rangle \in \mathsf{s}_1$. $b_d(\langle \mathsf{di}, \mathsf{rw}, \mathsf{l}\rangle) = \langle \mathsf{di}, \mathsf{rw}, b_m(\mathsf{l})\rangle \in \mathsf{s}_2$.

We now examine the possibilities when two equivalent program states take transitions. We first show that if one state takes a transition, then so does the other.

Lemma 4 *If* pt_1 *is consistent,* pt_2 *is consistent,* $\mathsf{pt}_1 \equiv_p \mathsf{pt}_2$ *and* $\mathsf{pt}_1 \rightarrow_p \mathsf{pt}'_1$ *then* $\exists \mathsf{pt}'_2 . \mathsf{pt}_2 \rightarrow_p \mathsf{pt}'_2$.

Proof Sketch: A case analysis of the axiom that generated $\mathsf{pt}_1 \rightarrow_p \mathsf{pt}'_1$ reveals that pt_2 can always take a corresponding transition generated by the same axiom.

We now characterize what can happen when equivalent states both take a transition. They key result is that two different transitions from equivalent parallel program states commute.

Lemma 5 *If* pt_1 *is consistent,* pt_2 *is consistent,* $\mathsf{pt}_1 \equiv_p \mathsf{pt}_2$, $\mathsf{pt}_1 \rightarrow_p \mathsf{pt}'_1$ *and* $\mathsf{pt}_2 \rightarrow_p \mathsf{pt}'_2$ *then either*

1. $\mathsf{pt}'_1 \equiv_p \mathsf{pt}'_2$ *or*
2. $\mathsf{pt}'_1 = \mathsf{pt}'_2$ *or*
3. $\mathsf{pt}'_1 = \mathtt{error}$, $\mathsf{pt}'_2 \rightarrow_p \mathtt{error}$ *or*
4. $\mathsf{pt}'_2 = \mathtt{error}$, $\mathsf{pt}'_1 \rightarrow_p \mathtt{error}$ *or*
5. $\exists \mathsf{pt}''_1, \mathsf{pt}''_2 . \mathsf{pt}'_1 \rightarrow_p \mathsf{pt}''_1, \mathsf{pt}'_2 \rightarrow_p \mathsf{pt}''_2$ *and* $\mathsf{pt}''_1 \equiv_p \mathsf{pt}''_2$.

Proof Sketch: Equivalent states have isomorphic sets of active tasks. If pt_1 and pt_2 took transitions from isomorphic tasks, the two program states pt'_1 and pt'_2 are equivalent. If one of the tasks generated a transition to `error`, then the isomorphic task can also generate that transition. If the two states both generated an integer result, then the definition of equivalence ensures that the results are the same.

For case 5, the transitions came from non-isomorphic tasks. An inspection of the axioms that define $\rightarrow_p$ reveals that all of the active tasks in pt_1 and pt_2 (with the possible exception of the tasks that generated the transitions) are still active in pt'_1 and pt'_2. Therefore, there is an active task in pt'_1 that is isomorphic to the task that generated $\mathsf{pt}_2 \rightarrow_p \mathsf{pt}'_2$ and vice-versa. These active tasks generate transitions $\mathsf{pt}'_1 \rightarrow_p \mathsf{pt}''_1$ and $\mathsf{pt}'_2 \rightarrow_p \mathsf{pt}''_2$.

We go through a case analysis of the possible pairs of transitions to show that $\mathsf{pt}''_1 \equiv_p \mathsf{pt}''_2$. In effect, we must show that the two transitions $\mathsf{pt}_1 \rightarrow_p \mathsf{pt}'_1$ and $\mathsf{pt}'_1 \rightarrow_p \mathsf{pt}''_1$ commute. The proof of commutativity relies on the definitions of consistency and equivalence of parallel program states.

We first show that the accesses to shared objects of any two active tasks do not conflict. For an active task to write a shared object, it must declare an immediate write on that shared object. By property 3 of definition 12 that declaration must be at the front of its queue. Therefore, no other task's read or write declaration can be at the front of the queue, and again by property 3 of definition 12 all other tasks that declare an immediate access on that object must be suspended. No other active task can access the object. The effects of the two transitions on the shared memory do not interfere, and therefore commute.

We must also verify that queue operations carried out by two distinct tasks commute. Each task changes some subset of the program state's queues. If the tasks change disjoint subsets, then their queue operations obviously commute. If the tasks change some of the same queues, then their aggregate queue operations commute if the operations commute for every queue. We therefore show that for any one queue, the two operations commute. If the two operations both modify the same queue, then the two tasks both have sets of declarations in the queue. By definition 8 the sets are disjoint and one set comes before the other with respect to $\sqsubset$. Queue removes and updates affect only each task's declarations; because the two sets of declarations are disjoint the operations commute. Inserts put a new set of declarations into the queue just before the parent task's declaration set; again because the two tasks' declarations sets are disjoint the operations commute.

We next show that if any parallel execution of a Jade program terminates without an error, then all parallel executions terminate and yield the same result.

Theorem 3
If $\mathsf{v} \in \mathsf{PObs}(c)$ *and* $\mathsf{v} \in \mathcal{Z}$*, then* $\mathsf{PObs}(c) = \{\mathsf{v}\}$*.*

Proof Sketch: An induction on the length of the reduction sequence using lemmas 2, 4 and 5.

Together theorems 2 and 3 establish the correspondence between the serial and parallel semantics. Theorem 2 says that if the serial execution of the program successfully halts, then at least one of the parallel executions successfully halts with the same result. Theorem 3 says that if one of the parallel executions successfully halts, then all of parallel executions successfully halt with the same result. So, a parallel execution of a Jade program will successfully halt if and only if the serial program successfully halts, and all such parallel and serial executions generate the same result.

The difference between the parallel semantics and the serial semantics is that the parallel semantics may terminate in the `error` state when the serial semantics does not terminate, and vice-versa. This can happen if the program has two independent tasks t_1 and t_2 such that t_1 runs forever and t_2 has an error. The serial semantics will always execute one of the two tasks before the other, and will therefore always get the same result. The parallel semantics, however, can execute the two tasks in parallel, producing either an infinite loop or an error depending on how the tasks' transitions interleave.

4 Comparison with Other Work

Explicitly parallel programming languages such as CSP [4], Ada [9], Linda [2], Occam [5] and ConcurrentSmalltalk [12] force the programmer to manage concurrency using low-level operations to synchronize parallel tasks. This explicitly parallel approach often leads to complicated, nondeterministic programs that are difficult to debug and maintain. Jade, on the other hand, adopts an *implicitly* parallel approach that maintains the programming advantages of serial languages. In [6] we present a detailed analysis of the differences between Jade and such explicitly parallel languages.

FX-87 [8] is similar to Jade in that it contains constructs that allow the programmer to express how the program accesses data. In FX-87, memory locations are partitioned into a finite, statically determined set of regions. The programmer declares the regions of memory that a function touches as part of the function's type. The FX-87 compiler can then verify the correspondence between the declared and actual data accesses, scheduling conflict-free pieces of the program for concurrent execution.

Making regions a static concept severely limits the amount of concurrency the implementation can extract [3]. At run time, multiple dynamic objects must be mapped to the same static region. Therefore, the compiler cannot exploit concurrency available between parts of the program that access disjoint sets of objects from the same region.

The other major difference between Jade and FX-87 is that FX-87 has no constructs for identifying task boundaries and synchronization points. It is the FX-87 compiler's responsibility to partition the program into tasks. We are aware of no generally applicable algorithm for successfully partitioning programs containing side effects.

5 Conclusion

Jade programmers implicitly express parallelism by specifying how a serial, imperative program uses data. Such an approach simplifies parallel programming and makes programs more portable across different parallel architectures.

The access specification is the key interface between the Jade programmer and the Jade implementation. The programmer uses Jade constructs to generate an access specification for every task. It is the programmer's responsibility to ensure that the access specification declares all of the task's accesses. Access specifications therefore restrict how parts of the program can access data. Based on these restrictions, the Jade implementation identifies tasks whose accesses do not conflict and executes these tasks concurrently. Finally, to detect data races caused by incorrect access specifications, the Jade implementation dynamically checks that each access specification is correct.

In this paper we present both a sequential and a parallel operational semantics; these semantics formally define the meaning of Jade programs. The proof of correspondence between the sequential and parallel operational semantics demonstrates that a Jade programmer can reason about parallel programs using a simple sequential programming model.

References

[1] M. Berry et al. The perfect club benchmarks: Effective performance evaluation of supercomputers. *International Journal of Supercomputer Applications*, 3(3):5–40, 1989.

[2] N. Carriero and D. Gelernter. How to Write Parallel Programs: A Guide to the Perplexed. *ACM Computing Surveys*, 21(3):323–357, September 1989.

[3] R. T. Hammel and D. K. Gifford. FX-87 Performance Measurements: Dataflow Implementation. Technical Report MIT/LCS/TR-421, MIT, November 1988.

[4] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, N.J., 1985.

[5] Inmos Ltd. *Occam Programming Manual*. Prentice-Hall, Englewood Cliffs, N.J., 1984.

[6] M. S. Lam and M. C. Rinard. Coarse-grain parallel programming in Jade. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 94–105, April 1991.

[7] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. L. Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 94–105, May 1990.

[8] J. M. Lucassen. Types and Effects: Towards the Integration of Functional and Imperative Programming. Technical Report MIT/LCS/TR-408, MIT, August 1987.

[9] United States Department of Defense. *Reference Manual for the Ada programming language*. DoD, Washington, D.C., January 1983. ANSI/MIL-STD-1815A.

[10] J. S. Rose. LocusRoute: A Parallel Global Router for Standard Cells. In *Proceedings of the 25th Design Automation Conference*, pages 189–195, June 1988.

[11] E. Rothberg and A. Gupta. Efficient sparse matrix factorization on high-performance workstations - exploiting the memory hierarchy. To appear in ACM Transactions on Mathematical Software.

[12] Y. Yokote and M. Tokoro. Concurrent Programming in ConcurrentSmalltalk. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 129–158. MIT Press, Cambridge, MA, 1987.

A Abstract Syntax

This section contains the abstract syntax for the simple sequential imperative language Simple. For purposes of exposition we use the access specification operations `im_rd` and `im_wr` instead of `rd` and `wr`.

$$
\begin{array}{rcll}
i & \in & \mathcal{Z} & \\
id & \in & \mathit{Id} & \\
c & \in & \mathit{Prog} & ::= \mathit{Code};\mathtt{result}(\mathit{Exp}) \\
c & \in & \mathit{Code} & ::= \mathit{Stmt};\mathit{Code} \mid \mathit{Jst};\mathit{Code} \mid \epsilon \\
st & \in & \mathit{Stmt} & ::= \mathit{Spec}(\mathit{Exp}) \mid \mathit{Id} := \mathit{Exp} \mid \\
 & & & \quad \mathit{Id} := \mathtt{pr}(\mathit{Exp}) \mid \mathit{Id} := \mathtt{sh}(\mathit{Exp}) \mid \\
 & & & \quad *\mathit{Exp} := \mathit{Exp} \mid \mathtt{while}\ (\mathit{Exp})\ \{\mathit{Code}\} \mid \\
 & & & \quad \mathtt{if}\ (\mathit{Exp})\ \{\mathit{Code}\}\ \mathtt{else}\ \{\mathit{Code}\} \mid \\
jst & \in & \mathit{Jst} & ::= \mathtt{with}\ \{\mathit{Code}\}\mathtt{cont} \mid \\
 & & & \quad \mathtt{withonly}\ \{\mathit{Code}\}\mathtt{do}(\mathit{Id},\ldots,\mathit{Id})\{\mathit{Code}\} \\
sp & \in & \mathit{Spec} & ::= \mathtt{im\_rd} \mid \mathtt{im\_wr} \mid \\
 & & & \quad \mathtt{df\_rd} \mid \mathtt{df\_wr} \mid \mathtt{no\_rd} \mid \mathtt{no\_wr} \\
exp & \in & \mathit{Exp} & ::= \mathcal{Z} \mid \mathit{Id} \mid \mathit{Exp}\ \mathit{Op}\ \mathit{Exp} \mid *\mathit{Exp} \mid \\
 & & & \quad \mathtt{is\_pr}(\mathit{Exp}) \mid \mathtt{is\_sh}(\mathit{Exp}) \\
op & \in & \mathit{Op} & ::= + \mid - \mid \times \mid = \mid < \mid >
\end{array}
$$

B Expression Evaluation

We define expression evaluation with the following axioms.

$$\frac{i \in \mathcal{Z}}{i\ \mathbf{in}\ \langle \mathsf{e}, \mathsf{m}, \mathsf{n}, \mathsf{s}\rangle = i}$$

$$\frac{id \in \mathrm{Dom}\ \mathsf{e}}{id\ \mathbf{in}\ \langle \mathsf{e}, \mathsf{m}, \mathsf{n}, \mathsf{s}\rangle = \mathsf{e}(id)}$$

$$\frac{exp\ \mathbf{in}\ \langle \mathsf{e}, \mathsf{m}, \mathsf{n}, \mathsf{s}\rangle = \mathsf{l} \in \mathrm{Dom}\ \mathsf{n}}{*exp\ \mathbf{in}\ \langle \mathsf{e}, \mathsf{m}, \mathsf{n}, \mathsf{s}\rangle = \mathsf{n}(\mathsf{l})}$$

$$\frac{exp \textbf{ in } \langle \mathsf{e},\mathsf{m},\mathsf{n},\mathsf{s}\rangle = \mathsf{l} \in \mathrm{Dom}\ \mathsf{m},\ \langle \mathtt{im},\mathtt{rd},\mathsf{l}\rangle \in \mathsf{s}}{*exp \textbf{ in } \langle \mathsf{e},\mathsf{m},\mathsf{n},\mathsf{s}\rangle = \mathsf{m}(\mathsf{l})}$$

$$\frac{exp_1 \textbf{ in } \langle \mathsf{e},\mathsf{m},\mathsf{n},\mathsf{s}\rangle = \mathsf{v}_1 \in \mathcal{Z},\ exp_2 \textbf{ in } \langle \mathsf{e},\mathsf{m},\mathsf{n},\mathsf{s}\rangle = \mathsf{v}_2 \in \mathcal{Z}}{exp_1\ op\ exp_2 \textbf{ in } \langle \mathsf{e},\mathsf{m},\mathsf{n},\mathsf{s}\rangle = \mathsf{v}_1\ op\ \mathsf{v}_2}$$

$$\frac{exp \textbf{ in } \langle \mathsf{e},\mathsf{m},\mathsf{n},\mathsf{s}\rangle = \mathsf{l} \in \mathrm{Dom}\ \mathsf{n}}{\mathtt{is\_pr}(exp) \textbf{ in } \langle \mathsf{e},\mathsf{m},\mathsf{n},\mathsf{s}\rangle = \mathtt{1},\ \mathtt{is\_sh}(exp) \textbf{ in } \langle \mathsf{e},\mathsf{m},\mathsf{n},\mathsf{s}\rangle = \mathtt{0}}$$

$$\frac{exp \textbf{ in } \langle \mathsf{e},\mathsf{m},\mathsf{n},\mathsf{s}\rangle = \mathsf{l} \in \mathrm{Dom}\ \mathsf{m}}{\mathtt{is\_pr}(exp) \textbf{ in } \langle \mathsf{e},\mathsf{m},\mathsf{n},\mathsf{s}\rangle = \mathtt{0},\ \mathtt{is\_sh}(exp) \textbf{ in } \langle \mathsf{e},\mathsf{m},\mathsf{n},\mathsf{s}\rangle = \mathtt{1}}$$

$$\frac{exp \textbf{ in } \langle \mathsf{e},\mathsf{m},\mathsf{n},\mathsf{s}\rangle = i \in \mathcal{Z}}{\mathtt{is\_pr}(exp) \textbf{ in } \langle \mathsf{e},\mathsf{m},\mathsf{n},\mathsf{s}\rangle = \mathtt{0},\ \mathtt{is\_sh}(exp) \textbf{ in } \langle \mathsf{e},\mathsf{m},\mathsf{n},\mathsf{s}\rangle = \mathtt{0}}$$

These axioms may leave an expression's value undefined if the expression or one of its subexpressions fails to satisfy the preconditions of one of the axioms. In this case the expression evaluates to the special value `error` indicating an evaluation error.

C Simple Transition Relation

$\rightarrow$ is defined to be the smallest relation satisfying the following axioms.

$$\frac{exp \textbf{ in } \langle \mathsf{e},\mathsf{m},\mathsf{n},\mathsf{s}\rangle = \mathsf{v} \in \mathsf{PrObj}}{id := exp;c \textbf{ in } \langle \mathsf{e},\mathsf{m},\mathsf{n},\mathsf{s}\rangle \rightarrow c \textbf{ in } \langle \mathsf{e}[id \mapsto \mathsf{v}],\mathsf{m},\mathsf{n},\mathsf{s}\rangle}$$

$$\frac{exp_1 \textbf{ in } \langle \mathsf{e},\mathsf{m},\mathsf{n},\mathsf{s}\rangle = \mathsf{l} \in \mathrm{Dom}\ \mathsf{n},\ exp_2 \textbf{ in } \langle \mathsf{e},\mathsf{m},\mathsf{n},\mathsf{s}\rangle = \mathsf{v} \in \mathsf{PrObj}}{*exp_1 := exp_2;c \textbf{ in } \langle \mathsf{e},\mathsf{m},\mathsf{n},\mathsf{s}\rangle \rightarrow c \textbf{ in } \langle \mathsf{e},\mathsf{m},\mathsf{n}[\mathsf{l} \mapsto \mathsf{v}],\mathsf{s}\rangle}$$

$$\frac{exp_1 \textbf{ in } \langle \mathsf{e},\mathsf{m},\mathsf{n},\mathsf{s}\rangle = \mathsf{l} \in \mathrm{Dom}\ \mathsf{m},\ \langle \mathtt{im},\mathtt{wr},\mathsf{l}\rangle \in \mathsf{s},\ exp_2 \textbf{ in } \langle \mathsf{e},\mathsf{m},\mathsf{n},\mathsf{s}\rangle = \mathsf{v} \in \mathsf{ShObj}}{*exp_1 := exp_2;c \textbf{ in } \langle \mathsf{e},\mathsf{m},\mathsf{n},\mathsf{s}\rangle \rightarrow c \textbf{ in } \langle \mathsf{e},\mathsf{m}[\mathsf{l} \mapsto \mathsf{v}],\mathsf{n},\mathsf{s}\rangle}$$

$$\frac{exp_1 \textbf{ in } \langle \mathsf{e},\mathsf{m},\mathsf{n},\mathsf{s}\rangle = \mathsf{l} \in \mathrm{Dom}\ \mathsf{m},\ \langle \mathtt{im},\mathtt{wr},\mathsf{l}\rangle \notin \mathsf{s}}{*exp_1 := exp_2;c \textbf{ in } \langle \mathsf{e},\mathsf{m},\mathsf{n},\mathsf{s}\rangle \rightarrow \mathtt{error}}$$

$$\frac{\mathsf{l} \in \mathsf{PrLoc} \setminus \mathrm{Dom}\ \mathsf{n},\ exp \textbf{ in } \langle \mathsf{e},\mathsf{m},\mathsf{n},\mathsf{s}\rangle = \mathsf{v} \in \mathsf{PrObj}}{id := \mathtt{pr}(exp);c \textbf{ in } \langle \mathsf{e},\mathsf{m},\mathsf{n},\mathsf{s}\rangle \rightarrow c \textbf{ in } \langle \mathsf{e}[id \mapsto \mathsf{l}],\mathsf{m},\mathsf{n}[\mathsf{l} \mapsto \mathsf{v}],\mathsf{s}\rangle}$$

$$\frac{\mathsf{l} \in \mathsf{ShLoc} \setminus \mathrm{Dom}\ \mathsf{m},\ exp \textbf{ in } \langle \mathsf{e},\mathsf{m},\mathsf{n},\mathsf{s}\rangle = \mathsf{v} \in \mathsf{ShObj},\ \mathsf{s}' = \mathsf{s} \cup \{\langle \mathtt{df},\mathtt{rd},\mathsf{l}\rangle, \langle \mathtt{df},\mathtt{wr},\mathsf{l}\rangle\}}{id := \mathtt{sh}(exp);c \textbf{ in } \langle \mathsf{e},\mathsf{m},\mathsf{n},\mathsf{s}\rangle \rightarrow c \textbf{ in } \langle \mathsf{e}[id \mapsto \mathsf{l}],\mathsf{m}[\mathsf{l} \mapsto \mathsf{v}],\mathsf{n},\mathsf{s}'\rangle}$$

$$\frac{exp \textbf{ in } \langle \mathsf{e},\mathsf{m},\mathsf{n},\mathsf{s}\rangle = \mathtt{1}}{\mathtt{if}\ (exp)\ \{c_1\}\ \mathtt{else}\ \{c_2\};c_3 \textbf{ in } \langle \mathsf{e},\mathsf{m},\mathsf{n},\mathsf{s}\rangle \rightarrow c_1;c_3 \textbf{ in } \langle \mathsf{e},\mathsf{m},\mathsf{n},\mathsf{s}\rangle}$$

$$\frac{exp \textbf{ in } \langle \mathsf{e},\mathsf{m},\mathsf{n},\mathsf{s}\rangle = \mathtt{0}}{\mathtt{if}\ (exp)\ \{c_1\}\ \mathtt{else}\ \{c_2\};c_3 \textbf{ in } \langle \mathsf{e},\mathsf{m},\mathsf{n},\mathsf{s}\rangle \rightarrow c_2;c_3 \textbf{ in } \langle \mathsf{e},\mathsf{m},\mathsf{n},\mathsf{s}\rangle}$$

$$\frac{exp \textbf{ in } \langle \mathsf{e},\mathsf{m},\mathsf{n},\mathsf{s}\rangle = \mathtt{1}}{\mathtt{while}\ (exp)\ \{c_1\};c_2 \textbf{ in } \langle \mathsf{e},\mathsf{m},\mathsf{n},\mathsf{s}\rangle \rightarrow c_1;\mathtt{while}\ (exp)\ \{c_1\};c_2 \textbf{ in } \langle \mathsf{e},\mathsf{m},\mathsf{n},\mathsf{s}\rangle}$$

$$\frac{exp \textbf{ in } \langle \mathsf{e},\mathsf{m},\mathsf{n},\mathsf{s}\rangle = \mathtt{0}}{\mathtt{while}\ (exp)\ \{c_1\};c_2 \textbf{ in } \langle \mathsf{e},\mathsf{m},\mathsf{n},\mathsf{s}\rangle \rightarrow c_2 \textbf{ in } \langle \mathsf{e},\mathsf{m},\mathsf{n},\mathsf{s}\rangle}$$

Definition 15 $\mathsf{subexp}(exp, st)$ *is true if exp appears in an expression of st.*

$$\frac{exp \textbf{ in } \langle \mathsf{e},\mathsf{m},\mathsf{n},\mathsf{s}\rangle = \mathtt{error},\ \mathsf{subexp}(exp, st)}{st;c_2 \textbf{ in } \langle \mathsf{e},\mathsf{m},\mathsf{n},\mathsf{s}\rangle \rightarrow \mathtt{error}}$$

D Jade Transition Relation

$\rightarrow_j$ is defined to be the smallest relation satisfying the following axioms.

$$\frac{c \textbf{ in } \langle \mathsf{e},\mathsf{m},\mathsf{n},\mathsf{s}\rangle \rightarrow c' \textbf{ in } \langle \mathsf{e}',\mathsf{m}',\mathsf{n}',\mathsf{s}'\rangle}{c \textbf{ in } \langle \mathsf{e},\mathsf{m},\mathsf{n},\mathsf{s},\mathsf{r}\rangle \rightarrow_j c' \textbf{ in } \langle \mathsf{e}',\mathsf{m}',\mathsf{n}',\mathsf{s}',\mathsf{r}\rangle}$$

$$\frac{c \textbf{ in } \langle \mathsf{e},\mathsf{m},\mathsf{n},\mathsf{s}\rangle \rightarrow \mathtt{error}}{c \textbf{ in } \langle \mathsf{e},\mathsf{m},\mathsf{n},\mathsf{s},\mathsf{r}\rangle \rightarrow_j \mathtt{error}}$$

$$\frac{c_1 \textbf{ in } \langle \mathsf{e},\mathsf{m},\mathsf{n},\mathsf{s}\rangle \rightarrow c_1' \textbf{ in } \langle \mathsf{e}',\mathsf{m}',\mathsf{n}',\mathsf{s}'\rangle}{\mathtt{withonly}\ \{c_1\}\mathtt{do}(ids)\{c_2\};c_3 \textbf{ in } \langle \mathsf{e},\mathsf{m},\mathsf{n},\mathsf{s},\mathsf{r}\rangle \rightarrow_j \mathtt{withonly}\ \{c_1'\}\mathtt{do}(ids)\{c_2\};c_3 \textbf{ in } \langle \mathsf{e}',\mathsf{m}',\mathsf{n}',\mathsf{s}',\mathsf{r}\rangle}$$

$$\frac{c_1 \textbf{ in } \langle \mathsf{e},\mathsf{m},\mathsf{n},\mathsf{s}\rangle \rightarrow c_1' \textbf{ in } \langle \mathsf{e}',\mathsf{m}',\mathsf{n}',\mathsf{s}'\rangle}{\mathtt{with}\ \{c_1\}\mathtt{cont};c_2 \textbf{ in } \langle \mathsf{e},\mathsf{m},\mathsf{n},\mathsf{s},\mathsf{r}\rangle \rightarrow_j \mathtt{with}\ \{c_1'\}\mathtt{cont};c_2 \textbf{ in } \langle \mathsf{e}',\mathsf{m}',\mathsf{n}',\mathsf{s}',\mathsf{r}\rangle}$$

$$\frac{c = \mathtt{withonly}\ \{\mathsf{di\_rw}(exp);c_1\}\mathtt{do}(ids)\{c_2\};c_3,\ exp \textbf{ in } \langle \mathsf{e},\mathsf{m},\mathsf{n},\mathsf{s}\rangle = \mathsf{l} \in \mathrm{Dom}\ \mathsf{m},\ \mathsf{r}' = \mathsf{r} \cup \{\langle \mathsf{di},\mathsf{rw},\mathsf{l}\rangle\}, \mathsf{s} \vdash \langle \mathsf{di},\mathsf{rw},\mathsf{l}\rangle}{c \textbf{ in } \langle \mathsf{e},\mathsf{m},\mathsf{n},\mathsf{s},\mathsf{r}\rangle \rightarrow_j \mathtt{withonly}\ \{c_1\}\mathtt{do}(ids)\{c_2\};c_3 \textbf{ in } \langle \mathsf{e},\mathsf{m},\mathsf{n},\mathsf{s},\mathsf{r}'\rangle}$$

$$\frac{exp \textbf{ in } \langle \mathsf{e},\mathsf{m},\mathsf{n},\mathsf{s}\rangle = \mathsf{l} \in \mathrm{Dom}\ \mathsf{m}, \mathsf{s} \vdash \langle \mathsf{di},\mathsf{rw},\mathsf{l}\rangle}{\mathtt{with}\ \{\mathsf{di\_rw}(exp);c_1\}\mathtt{cont};c_2 \textbf{ in } \langle \mathsf{e},\mathsf{m},\mathsf{n},\mathsf{s},\mathsf{r}\rangle \rightarrow_j \mathtt{with}\ \{c_1\}\mathtt{cont};c_2 \textbf{ in } \langle \mathsf{e},\mathsf{m},\mathsf{n},\mathsf{s},\mathsf{r} \cup \{\langle \mathsf{di},\mathsf{rw},\mathsf{l}\rangle\}\rangle}$$

$$\frac{}{\langle \mathsf{m}, \langle \mathsf{di\_rw}(exp);c,\mathsf{e},\mathsf{n},\mathsf{s},\mathsf{r}\rangle \circ \mathsf{ts}\rangle \rightarrow_j \mathtt{error}}$$

Definition 16 $\langle \mathsf{e}_1,\mathsf{m}_1,\mathsf{n}_1,\mathsf{s}_1,\mathsf{r}_1\rangle \equiv_j^b \langle \mathsf{e}_2,\mathsf{m}_2,\mathsf{n}_2,\mathsf{s}_2,\mathsf{r}_2\rangle$ *iff* $\langle \mathsf{e}_1,\mathsf{m}_1,\mathsf{n}_1,\mathsf{s}_1\rangle \equiv^b \langle \mathsf{e}_2,\mathsf{m}_2,\mathsf{n}_2,\mathsf{s}_2\rangle$ *and* $\langle \mathsf{e}_1,\mathsf{m}_1,\mathsf{n}_1,\mathsf{r}_1\rangle \equiv^b \langle \mathsf{e}_2,\mathsf{m}_2,\mathsf{n}_2,\mathsf{r}_2\rangle$

E Serial Transition Relation

$\rightarrow_s$ is defined to be the smallest relation satisfying the following axioms.

$$\frac{c \textbf{ in } \langle \mathsf{e}, \mathsf{m}, \mathsf{n}, \mathsf{s}, \mathsf{r} \rangle \rightarrow_j c' \textbf{ in } \langle \mathsf{e}', \mathsf{m}', \mathsf{n}', \mathsf{s}', \mathsf{r}' \rangle}{\langle \mathsf{m}, \langle c, \mathsf{e}, \mathsf{n}, \mathsf{s}, \mathsf{r} \rangle \circ \mathsf{ts} \rangle \rightarrow_s \langle \mathsf{m}', \langle c', \mathsf{e}', \mathsf{n}', \mathsf{s}', \mathsf{r}' \rangle \circ \mathsf{ts} \rangle}$$

$$\frac{c \textbf{ in } \langle \mathsf{e}, \mathsf{m}, \mathsf{n}, \mathsf{s}, \mathsf{r} \rangle \rightarrow_j \texttt{error}}{\langle \mathsf{m}, \langle c, \mathsf{e}, \mathsf{n}, \mathsf{s}, \mathsf{r} \rangle \circ \mathsf{ts} \rangle \rightarrow_s \texttt{error}}$$

$$\frac{c = \texttt{with } \{\epsilon\} \texttt{cont}; c'}{\langle \mathsf{m}, \langle c, \mathsf{e}, \mathsf{n}, \mathsf{s}, \mathsf{r} \rangle \rangle \rightarrow_s \langle \mathsf{m}, \langle c', \mathsf{e}, \mathsf{n}, \mathsf{s} \uparrow \mathsf{r}, \emptyset, \rangle \rangle}$$

$$\frac{\begin{array}{c} c = \texttt{withonly } \{\epsilon\} \mathsf{do}(id_1, \ldots, id_n)\{c_1\}; c_2, \\ \forall i \leq n. id_i \in \text{Dom } \mathsf{e} \text{ and } \mathsf{e}(id_i) \notin \mathsf{PrLoc}, \\ \mathsf{e}' = [id_1 \mapsto \mathsf{e}(id_1)] \cdots [id_n \mapsto \mathsf{e}(id_n)] \end{array}}{\begin{array}{c} \langle \mathsf{m}, \langle c, \mathsf{e}, \mathsf{n}, \mathsf{s}, \mathsf{r} \rangle \circ \mathsf{ts} \rangle \rightarrow_s \\ \langle \mathsf{m}, \langle c_1, \mathsf{e}', \emptyset, \emptyset \uparrow \mathsf{r}, \emptyset \rangle \circ \langle c_2, \mathsf{e}, \mathsf{n}, \mathsf{s}, \emptyset \rangle \circ \mathsf{ts} \rangle \end{array}}$$

$$\frac{}{\langle \mathsf{m}, \langle \epsilon, \mathsf{e}, \mathsf{n}, \mathsf{s}, \mathsf{r} \rangle \circ \mathsf{ts} \rangle \rightarrow_s \langle \mathsf{m}, \mathsf{ts} \rangle}$$

$$\frac{c = \texttt{result}(exp), exp \textbf{ in } \langle \mathsf{e}, \mathsf{m}, \mathsf{n}, \mathsf{s} \rangle = \mathsf{v} \in \mathcal{Z}}{\langle \mathsf{m}, \langle c, \mathsf{e}, \mathsf{n}, \mathsf{s}, \mathsf{r} \rangle \rangle \rightarrow_s \mathsf{v}}$$

F Parallel Transition Relation

$\rightarrow_p$ is defined to be the smallest relation satisfying the following axioms.

$$\frac{\begin{array}{c} \mathsf{t} = \langle c, \mathsf{e}, \mathsf{n}, \mathsf{s}, \mathsf{r} \rangle \in \mathsf{A}, \\ c \textbf{ in } \langle \mathsf{e}, \mathsf{m}, \mathsf{n}, \mathsf{s}, \mathsf{r} \rangle \rightarrow_j c' \textbf{ in } \langle \mathsf{e}', \mathsf{m}', \mathsf{n}', \mathsf{s}', \mathsf{r}' \rangle \end{array}}{\begin{array}{c} \langle \mathsf{m}, \mathsf{A}, \mathsf{S}, \sqsubset \rangle \rightarrow_p \\ \langle \mathsf{m}', \mathsf{A} \setminus \{\mathsf{t}\} \uplus \{\langle c', \mathsf{e}', \mathsf{n}', \mathsf{s}', \mathsf{r}' \rangle\}, \mathsf{S}, \sqsubset \rangle \end{array}}$$

$$\frac{\mathsf{t} = \langle c, \mathsf{e}, \mathsf{n}, \mathsf{s}, \mathsf{r} \rangle \in \mathsf{A}, c \textbf{ in } \langle \mathsf{e}, \mathsf{m}, \mathsf{n}, \mathsf{s}, \mathsf{r} \rangle \rightarrow_j \texttt{error}}{\langle \mathsf{m}, \mathsf{A}, \mathsf{S}, \sqsubset \rangle \rightarrow_p \texttt{error}}$$

$$\frac{\begin{array}{c} \mathsf{t} = \langle \texttt{with } \{\epsilon\} \texttt{cont}; c, \mathsf{e}, \mathsf{n}, \mathsf{s}, \mathsf{r} \rangle \in \mathsf{A}, \\ \mathsf{t}' = \langle c, \mathsf{e}, \mathsf{n}, \mathsf{s} \uparrow \mathsf{r}, \emptyset \rangle \end{array}}{\begin{array}{c} \langle \mathsf{m}, \mathsf{A}, \mathsf{S}, \sqsubset \rangle \rightarrow_p \\ \langle \mathsf{m}, \mathsf{A} \setminus \{\mathsf{t}\}, \mathsf{S} \uplus \{\mathsf{t}'\}, \mathsf{upd}(\mathsf{s}, \mathsf{s} \uparrow \mathsf{r}, \sqsubset) \rangle \end{array}}$$

$$\frac{\begin{array}{c} c = \texttt{withonly } \{\epsilon\} \mathsf{do}(id_1, \ldots, id_n)\{c_1\}; c_2, \\ \mathsf{t} = \langle c, \mathsf{e}, \mathsf{n}, \mathsf{s}, \mathsf{r} \rangle \in \mathsf{A}, \\ \forall i \leq n. id_i \in \text{Dom } \mathsf{e} \text{ and } \mathsf{e}(id_i) \notin \mathsf{PrLoc} \\ \mathsf{e}' = [id_1 \mapsto \mathsf{e}(id_1)] \cdots [id_n \mapsto \mathsf{e}(id_n)], \\ \mathsf{t}' = \langle c_1, \mathsf{e}', \emptyset, \emptyset \uparrow \mathsf{r}, \emptyset \rangle, \mathsf{t}'' = \langle c_2, \mathsf{e}, \mathsf{n}, \mathsf{s}, \emptyset \rangle \end{array}}{\begin{array}{c} \langle \mathsf{m}, \mathsf{A}, \mathsf{S}, \sqsubset \rangle \rightarrow_p \\ \langle \mathsf{m}, \mathsf{A} \setminus \{\mathsf{t}\}, \mathsf{S} \uplus \{\mathsf{t}', \mathsf{t}''\}, \mathsf{ins}(\emptyset \uparrow \mathsf{r}, \mathsf{s}, \sqsubset) \rangle \end{array}}$$

$$\frac{\mathsf{t} = \langle \epsilon, \mathsf{e}, \mathsf{n}, \mathsf{s}, \mathsf{r} \rangle \in \mathsf{A},}{\langle \mathsf{m}, \mathsf{A}, \mathsf{S}, \sqsubset \rangle \rightarrow_p \langle \mathsf{m}, \mathsf{A} \setminus \{\mathsf{t}\}, \mathsf{S}, \mathsf{rem}(\mathsf{s}, \sqsubset) \rangle}$$

$$\frac{\begin{array}{c} \mathsf{t} = \langle c, \mathsf{e}, \mathsf{n}, \mathsf{s}, \mathsf{r} \rangle \in \mathsf{S}, \\ \forall \langle \texttt{im}, \mathsf{rw}, \mathsf{l} \rangle \in \mathsf{s.f}(\langle \texttt{im}, \mathsf{rw}, \mathsf{l} \rangle, \sqsubset) \end{array}}{\langle \mathsf{m}, \mathsf{A}, \mathsf{S}, \sqsubset \rangle \rightarrow_p \langle \mathsf{m}, \mathsf{A} \uplus \{\mathsf{t}\}, \mathsf{S} \setminus \{\mathsf{t}\}, \sqsubset \rangle}$$

$$\frac{\begin{array}{c} c = \texttt{result}(exp), \\ exp \textbf{ in } \langle \mathsf{e}, \mathsf{m}, \mathsf{n}, \mathsf{s} \rangle = \mathsf{v} \in \mathcal{Z} \end{array}}{\langle \mathsf{m}, \{\langle c, \mathsf{e}, \mathsf{n}, \mathsf{s}, \mathsf{r} \rangle\}, \emptyset, \sqsubset \rangle \rightarrow_p \mathsf{v}}$$