A Semantic Framework for Designer Transactions

Jan Vitek, Suresh Jagannathan, Adam Welc, and Antony L. Hosking

Purdue University

Abstract. A transaction defines a locus of computation that satisfies important concurrency and failure properties; these so-called ACID properties provide strong serialization guarantees that allow us to reason about concurrent and distributed programs in terms of higher-level units of computation (*e.g.*, transactions) rather than lower-level data structures (*e.g.*, mutual-exclusion locks). This paper presents a framework for specifying the semantics of a transactional facility integrated within a host programming language. The TFJ calculus supports nested and multi-threaded transactions. We give a semantics to TFJ that is parameterized by the definition of the transactional mechanism that permits the study of different transaction models.

1 Introduction

The integration of transactional facilities into programming languages has been driven by applications ranging from middleware infrastructure for enterprise applications to runtime support for optimistic concurrency. The concept of transactions is well-known in database systems; the main challenge in associating transactions with programming control structures comes from the mismatch between the concurrency model of the programming language and the concurrency model of the transactional facility. Existing technologies enforce little or no relationship between these models, so that programmers can neither rely on transactions for complete isolation among concurrent threads, nor use concurrent threading to more conveniently program transaction logic.

As a first step towards addressing some of these concerns, we propose a semantic framework in which different transactional mechanisms can be studied and compared formally. Requirements for such a framework are that it be sufficiently expressive to allow the specification of core transactional features, that it provide a way to validate the correctness of the semantics, and that it support features found in realistic programs. We are interested in the impact of design choices on observable behavior (*e.g.*, aborts, deadlocks, livelocks) and on implementation performance (*e.g.*, space and time overhead). Our long-term goal is to leverage this framework to aid in the definition of program analyzes and optimization techniques for transactional languages.

This paper introduces TFJ, or *Transactional Featherweight Java*, an object calculus with syntactic support for transactions. The operational semantics of TFJ is given in terms of a stratified set of rewrite rules parameterized over the meaning of the transactional constructs. This allows us to define a variety of transactional semantics within the same core language. In this paper, we study *nested and multi-threaded* transactions with two different concurrency control models (two-phase locking and versioning). The primary contribution of this paper is a formal characterization and proof of correctness of different transactional models when incorporated into the core language. While there has been significant previous work on devising formal notation and specifications [16,5] to describe transactional properties, we are not aware of other efforts that use operational semantics to study the interplay of concurrency and serializability among different transactional models.

2 The Transactional Featherweight Java Calculus

Transactional Featherweight Java (TFJ) is a new object calculus inspired by the work of Igarashi *et al.* [14]. TFJ includes threads and the imperative constructs needed to model transactions. In this paper, we focus on a simplified variant of TFJ, that is dynamically typed and in which all classes directly inherit a distinguished `Object` class. We introduce the syntax and semantics of TFJ with an example. Consider the classes given in Fig. 1. Class `Updater` encapsulates an update to an object. Class `Runner` is designed to perform an action within a new thread. Class `Transactor` performs two operations within a transaction. In this example, class `Updater` has two fields, `n` and `v`, and an `update()` method which assigns `v` to `n`'s `val` field. `Runner` has a `run()` method which starts a new thread and invokes a method on its `r` field within that thread. `Transactor` has a `pick()` method which is used to evaluate two expressions in a non-deterministic order; non-determinism is achieved since the order in which arguments are evaluated

```
class Updater {
   n, v;
   init( n, v) { return (this.n := n; this.v := v; this); }
   update() { return this.n.val:= this.v; } }

class Runner {
   s;
   init( s) { return (this.s := s; this); }
   run() { return spawn this.s.run(); } }

class Transactor {
   u, r;
   init( r, u) { return (this.u := u; this.r := r; this); }
   pick( _, _, v) { return v; }
   run() { return (
         onacid;
         this.pick( this.u.update(), this.r.run(), this.u.n.val);
         commit); } }
```

Fig. 1. An example TFJ program.

in a method call is unspecified. It also has a `run()` method which starts a new transaction and invokes `update` on field `u` and `run()` on field `r`. The keyword **onacid** marks the beginning of a transaction and **commit** ends the current transaction (as determined by the last **onacid**). All objects have an `init()` method which assigns values to their fields and returns `this`. Fig. 2 gives a TFJ code fragment making use of the above class definitions. Variable `n` is bound to a new object of some class `Number` (whose only feature of interest is that it must have a `val` field; we further assume the existence of classes `One`, `Two`, and `Three` that define specific numbers). `Noop` is an initialized `Runner` object with an uninteresting `run` method. Objects `l1` and `l2` are transactors which will be used to initiate nested transaction (`l2` within `l1`). Two runner objects will be used to create threads $\mathtt{t}_1$ and $\mathtt{t}_2$.

```
n := new Number();
s1 := new Transactor.init()( Noop, new Updater().init( n, new One()));
r1 := new Runner().init( s1);
s2 := new Transactor.init()( r1, new Updater().init( n, new Two()));
new Runner().init( s2).run();
n.val := new Three()
```

Fig. 2. A TFJ code fragement using definition of Fig. 1.

Evaluating the program of Fig. 2 will result in the creation of two threads ($\mathtt{t}_1$ and $\mathtt{t}_2$) and two new transactions ($\mathtt{l}_1$ and $\mathtt{l}_2$). Thread $\mathtt{t}_1$ executes solely within transaction $\mathtt{l}_1$, while $\mathtt{t}_2$ starts executing in $\mathtt{l}_1$, before starting transaction $\mathtt{l}_2$. We assume that there is a default top-level transaction, $\mathtt{l}_0$ and primordial thread $\mathtt{t}_0$. Fig. 3 shows the structure of this computation. The threads in a parent transaction can execute concurrently with threads in nested transactions. A design choice in TFJ is that all threads must join (via a commit) for the entire transaction to commit. Alternatives, such as interrupting threads that are not at a commit point when another thread in the same transaction is ready to commit, or silently committing changes while the thread is running are either programmer unfriendly or counter to the spirit of transactions.

The states in this program are defined completely by the instance of class `Number` that is threaded through transactions and handed down to `Updater`s for modi-

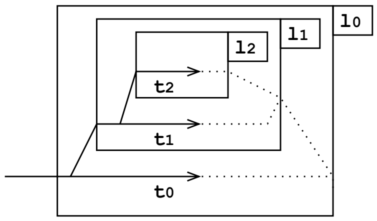

Fig. 3. Threads and transactions in Fig. 2.

fication. Each invocation of `update()` performs a read and a write of `val`. One valid interleaving of the operations is, for example:

$$[\mathtt{n} := \mathtt{One()}]_{\mathtt{l}_1} \;\rightarrow\; [\mathtt{n}]_{\mathtt{l}_1} \;\rightarrow\; [\mathtt{n} := \mathtt{Two()}]_{\mathtt{l}_2} \;\rightarrow\; [\mathtt{n}]_{\mathtt{l}_2} \;\rightarrow\; [\mathtt{n} := \mathtt{Three()}]_{\mathtt{l}_0}$$

This is correct because all of the changes performed by $\mathtt{l}_1$ occur before changes (reads and writes) performed by transactions $\mathtt{l}_2$ and $\mathtt{l}_0$. An invalid interleaving of these operations is:

$$[\mathtt{n} := \mathtt{One()}]_{\mathtt{l}_1} \;\rightarrow\; [\mathtt{n} := \mathtt{Two()}]_{\mathtt{l}_2} \;\rightarrow\; [\mathtt{n}]_{\mathtt{l}_1} \;\rightarrow\; [\mathtt{n}]_{\mathtt{l}_2} \;\rightarrow\; [\mathtt{n} := \mathtt{Three()}]_{\mathtt{l}_0}$$

In this schedule serializability is broken because $\mathtt{l}_1$ reads the value of `n.val` that was changed by $\mathtt{l}_2$. Thus from $\mathtt{l}_1$'s viewpoint the global state is $\mathtt{n.val} = \mathtt{Two}()$. Most concurrency control protocols will flag this as a conflict and abort $\mathtt{l}_1$. We note that in this particular case the conflict is benign as $\mathtt{l}_1$ discards the value it reads and thus the state of the system is not affected by it reading a stale value.

2.1 Syntax

The syntax of TFJ is given in Fig. 4. We take metavariables `L` to range over class declarations, $\mathtt{C}, \mathtt{D}$ to range over classes, `M` to range over methods, and `f` and `x` to range over fields and parameters, respectively. We also use `P` for process terms, `e` for expressions and $\mathtt{v}, \mathtt{u}$ for memory references. We use over-bar to represent a finite ordered sequence, for instance, $\overline{\mathtt{f}}$ represents $\mathtt{f}_1\,\mathtt{f}_2\,\ldots\,\mathtt{f}_\mathtt{n}$. The term $\overline{\mathtt{l}}\mathtt{l}$ denotes the extension of the sequence $\overline{\mathtt{l}}$ with a single element `l`, and $\overline{\mathtt{l}}\;.\;\overline{\mathtt{l}'}$ for sequence concatenation. We write $\overline{\mathtt{l}} \lhd \overline{\mathtt{l}}'$ if $\overline{\mathtt{l}}$ is a prefix of $\overline{\mathtt{l}}'$.

The calculus has a call-by-value semantics. The expression $\mathtt{C}(\overline{\mathtt{v}}) \downarrow_{\mathtt{i}}^{\mathtt{v}'}$ yields an object identical to $\mathtt{C}(\overline{\mathtt{v}})$ except in the ith field which is set to $\mathtt{v}'$. The `null` metavariable is used to represent an unbound reference. By default all objects are null initialized (*i.e.* $\mathtt{C}(\overline{\mathtt{null}})$).

Since TFJ has by-value semantics for invocation, sequencing can be encoded as a sequence of method invocations. For readability, we sometimes write "$(\mathtt{e}_1;\mathtt{e}_2)$" in examples to indicate sequencing of expressions $\mathtt{e}_1$ and $\mathtt{e}_2$. The value of a sequence is always the value of the last expression.

An *expression* `e` can be either a variable `x`, the `this` pseudo variable, a reference `v`, a field access `e.f`, a method invocation $\mathtt{e.m}(\overline{\mathtt{e}})$, an object construction `newC()`, a thread creation `spawn e`, an `onacid` command or a `commit`. The latter three operations are unique to TFJ. The expression `spawn e` creates a new thread of control to evaluate `e`. The evaluation of `e` takes place in the same environment as the thread executing `spawn e`. A new transaction is started by executing `onacid`. The dynamic context of `onacid` is delimited by `commit`. Effects performed within the context of `onacid` are not visible outside the transaction until a `commit` occurs. Transactions may be nested. When the `commit` of an inner transaction occurs, its effects are propagated to its parent. Threads may be spawned within the context of a transaction. The local state of the transaction is visible to all threads which execute within it. Transactions may also execute concurrently. For example, in `spawn e`, `e` may be an expression that includes `onacid` and `commit`; the transaction created by `onacid` executes concurrently with the thread executing the `spawn` operation. A process term $\mathtt{P}$ can be either the empty process $\mathtt{0}$, the parallel composition of process $P \mid P$ or a thread $\mathtt{t}$ running expression $\mathtt{e}$, denoted $\mathtt{t}[\,\mathtt{e}\,]$.

Note that the language does not provide an explicit `abort` operation. Transactions may abort *implicitly* because serialization invariants are violated. Our semantics expresses implicit aborts both in the definition of `commit` and in the treatment of read and write operations that would otherwise expose violations of necessary serializability invariants. Implicit aborts are tantamount to stuck states.

2.2 Reduction

The dynamic semantics of our language shown in Figs. 4 and 5 is given by a two-level set of rewrite rules. The computational core of the language is defined by a reduction relation of the from $\mathcal{E}\ \mathtt{e} \xrightarrow{\alpha} \mathcal{E}'\ \mathtt{e}'$. Here $\mathcal{E}$ is an environment containing bindings from references to objects ($\mathtt{v} \mapsto \mathtt{C}(\overline{\mathtt{v}})$), $\mathtt{e}$ is an expression and the action label α determines which reduction was picked. Action labels for the computational core are selected from the set $\{rd, wr, xt\}$, respectively read, write and extend. In addition to specifying the action on whose behalf a particular reduction is taken, we also specify the action's effects; for example, we write $wr\ vv'$to denote an action with label wr which has effect on locations v and v'. A read action effects the location being read, a write action has an effect on both the location being written and the location whose value it reads, and an extend operation has an effect on the newly created location.

A second reduction relation $\overset{\alpha}{\Longrightarrow}_{\mathtt{t}}$ defines operations over the entire program and has the form $\Gamma\ P \overset{\alpha}{\Longrightarrow}_{\mathtt{t}} \Gamma'\ P'$ where Γ is a program state composed of a sequence of thread environments $\overline{\mathtt{t}, \mathcal{E}}$ where each $\mathtt{t}, \mathcal{E}$ pair represents the association of a thread to its environment. The action label α can be one of the computational core labels or one of $\{sp, ac, co, ki\}$ for, respectively, spawn, onacid, commit, or kill.

The metavariable $\mathtt{l}$ ranges over transaction names, sequences of transaction names are used to represent the nesting structure. The transaction label $\overline{\mathtt{l}}$ identifies the transaction on whose behalf the reduction step was performed. As usual, $\overset{\alpha}{\Longrightarrow}_{\mathtt{t}} *$ denotes the reflexive and transitive closure of the global reduction relation. The congruence rules given in Figure 5 are straightforward.

We work up to congruence of processes ($\mathtt{P}|\mathtt{P}' \equiv \mathtt{P}'|\mathtt{P}$ and $\mathtt{P}|\mathtt{0} \equiv \mathtt{P}$). Congruence over expressions is defined in terms of evaluation contexts, $E[\,\bullet\,]$. The relations $\Downarrow_{\mathtt{spawn}}$, $\Downarrow_{\mathtt{onacid}}$ and $\Downarrow_{\mathtt{commit}}$ are used to extract nested expressions out of a context. The other definitions are similar to those used in the specification of FJ. *fields*

Syntax:

$$
\begin{array}{l}
\mathtt{P} ::= \mathtt{0} \mid \mathtt{P|P} \mid \mathtt{t[e]} \\
\mathtt{L} ::= \mathtt{class\ C}\ \{\ \overline{\mathtt{f}};\ \overline{\mathtt{M}}\ \} \\
\mathtt{M} ::= \mathtt{m}(\overline{\mathtt{x}})\ \{\ \mathtt{return\ e};\ \} \\
\mathtt{e} ::= \mathtt{x} \mid \mathtt{this} \mid \mathtt{v} \mid \mathtt{e.f} \mid \mathtt{e.m}(\overline{\mathtt{e}}) \mid \mathtt{e.f} := \mathtt{e} \mid \\
\qquad \mathtt{new\ C()} \mid \mathtt{spawn\ e} \mid \mathtt{onacid} \mid \mathtt{commit} \mid \mathtt{null}
\end{array}
$$

Lookup:

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C}\ \{\ \overline{\mathtt{f}};\ \overline{\mathtt{M}}\ \}}{\mathit{fields}(\mathtt{C}) = (\overline{\mathtt{f}})}$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C}\ \{\ \overline{\mathtt{f}};\ \overline{\mathtt{M}}\ \} \qquad \mathtt{m}(\overline{\mathtt{x}})\ \{\ \mathtt{return\ e};\ \} \in \overline{\mathtt{M}}}{\mathit{mbody}(\mathtt{m},\ \mathtt{C}) = (\overline{\mathtt{x}},\ \mathtt{e})}$$

Local Computation:

$$\frac{\mathcal{E}', \mathtt{C}(\overline{\mathtt{u}}) = \mathit{read}(\mathtt{v}, \mathcal{E}) \quad \mathit{fields}(\mathtt{C}) = (\overline{\mathtt{f}})}{\mathcal{E}\ \mathtt{v.f_i} \xrightarrow{rd\ \mathtt{v}} \mathcal{E}'\ \mathtt{u}_i} \qquad (\textsc{R-Field})$$

$$\frac{\mathcal{E}', \mathtt{C}(\overline{\mathtt{v}}) = \mathit{read}(\mathtt{v}, \mathcal{E}) \quad \mathcal{E}'' = \mathit{write}(\mathtt{v} \mapsto \mathtt{C}(\overline{\mathtt{v}})\downarrow^{\mathtt{v}'}_{\mathtt{i}}, \mathcal{E}')}{\mathcal{E}\ \mathtt{v.f_i} := \mathtt{v}' \xrightarrow{wr\ \mathtt{vv}'} \mathcal{E}''\ \mathtt{v}'} \qquad (\textsc{R-Assign})$$

$$\frac{\mathcal{E}', \mathtt{C}(\overline{\mathtt{u}}) = \mathit{read}(\mathtt{v}, \mathcal{E}) \quad \mathit{mbody}(\mathtt{m},\ \mathtt{C}_0) = (\overline{\mathtt{x}},\ \mathtt{e})}{\mathcal{E}\ \mathtt{v.m}(\overline{\mathtt{v}}) \xrightarrow{rd\ \mathtt{v}} \mathcal{E}'\ [\overline{\mathtt{v}}/\overline{\mathtt{x}}, \mathtt{v}/\mathtt{this}]\mathtt{e}} \qquad (\textsc{R-Invk})$$

$$\frac{\mathtt{v}\ \text{fresh} \quad \mathcal{E}' = \mathit{extend}(\mathtt{v} \mapsto \mathtt{C}(\overline{\mathtt{null}}), \mathcal{E})}{\mathcal{E}\ \mathtt{new\ C()} \xrightarrow{xt\ \mathtt{v}} \mathcal{E}'\ \mathtt{v}} \qquad (\textsc{R-New})$$

Global Computation:

$$\frac{\begin{array}{c} P = P'' \mid \mathtt{t[e]} \quad \mathcal{E}\ \mathtt{e} \xrightarrow{\alpha} \mathcal{E}'\ \mathtt{e}' \quad P' = P'' \mid \mathtt{t[e']} \\ \Gamma = \mathtt{t}, \mathcal{E}\ .\ \Gamma'' \quad \Gamma' = \mathit{reflect}(\mathtt{t},\ \mathcal{E}', \Gamma'') \quad \ell(\mathtt{t}, \Gamma) = \overline{\mathtt{l}} \end{array}}{\Gamma\ P \overset{\alpha}{\Longrightarrow}_{\mathtt{t}} \Gamma'\ P'} \qquad (\textsc{G-Plain})$$

$$\frac{\begin{array}{c} P = P'' \mid \mathtt{t[e]} \quad \mathtt{e} \Downarrow_{\mathtt{spawn}} \mathtt{e}', \mathtt{e}'' \quad P' = P'' \mid \mathtt{t[e']} \mid \mathtt{t'[e'']} \\ \mathtt{t}'\ \text{fresh} \quad \Gamma' = \mathit{spawn}(\mathtt{t}, \mathtt{t}', \Gamma) \quad \ell(\mathtt{t}, \Gamma) = \overline{\mathtt{l}} \end{array}}{\Gamma\ P \overset{sp\ \mathtt{t}'}{\Longrightarrow}_{\mathtt{t}} \Gamma'\ P'} \qquad (\textsc{G-Spawn})$$

$$\frac{\begin{array}{c} P = P'' \mid \mathtt{t[e]} \quad \mathtt{e} \Downarrow_{\mathtt{onacid}} \mathtt{e}' \quad P' = P'' \mid \mathtt{t[e']} \\ \mathtt{l}\ \text{fresh} \quad \Gamma' = \mathit{start}(\mathtt{l}, \mathtt{t}, \Gamma) \quad \ell(\mathtt{t}, \Gamma) = \overline{\mathtt{l}} \end{array}}{\Gamma\ P \overset{ac}{\Longrightarrow}_{\mathtt{t}} \Gamma'\ P'} \qquad (\textsc{G-Trans})$$

$$\frac{\begin{array}{c} P = P'' \mid \overline{\mathtt{t[e]}} \quad \overline{\mathtt{e}} \Downarrow_{\mathtt{commit}} \overline{\mathtt{e}}' \quad P' = P'' \mid \overline{\mathtt{t[e']}} \quad \overline{\mathtt{t}} = \mathit{intranse}(\mathtt{l}, \Gamma) \\ \Gamma = \mathtt{t}_0\ \mathcal{E}\ .\ \Gamma'' \quad \Gamma' = \mathit{commit}(\overline{\mathtt{t}}, \mathcal{E}, \Gamma) \quad \ell(\mathtt{t}', \Gamma) = \overline{\mathtt{l}} \end{array}}{\Gamma\ P \overset{co}{\Longrightarrow}_{\mathtt{t}'} \Gamma'\ P'} \qquad (\textsc{G-Comm})$$

$$\frac{P = P' \mid \mathtt{t[v]} \quad \Gamma = \mathtt{t}, \mathcal{E}\ .\ \Gamma' \quad \ell(\mathtt{t}, \Gamma) = \overline{\mathtt{l}}}{\Gamma\ P \overset{ki}{\Longrightarrow}_{\mathtt{t}} \Gamma'\ P'} \qquad (\textsc{G-ThKill})$$

Fig. 4. TFJ Syntax and Semantics.

returns the list of all fields of a class including inherited ones. *mbody* returns the body of the method in a given class.

Let $\mathcal{E}$ be an environment of the form $\mathtt{l}_0{:}\rho_0 \ldots \mathtt{l}_n{:}\rho_n$, then $\ell(\mathcal{E})$ extracts the order transaction label sequence, $\ell(\mathcal{E}) = \mathtt{l}_0 \ldots \mathtt{l}_n$ if $\Gamma = \mathtt{t}, \mathcal{E}\ .\ \Gamma'$ and $\ell(\mathtt{t}, (\mathtt{t}, \mathcal{E}\ .\ \Gamma)) = \ell(\mathcal{E})$. The auxiliary function $\mathit{last}(\mathtt{v}, \rho)$ is defined to return a one element sequence containing the last value referenced by $\mathtt{v}$ in the sequence of bindings ρ or the empty sequence if there is no binding for $\mathtt{v}$. It is defined inductively to return $\langle\,\rangle$ if $\rho = \langle\,\rangle$, $\mathtt{C}(\overline{\mathtt{v}})$ if $\rho = \rho' \,.\, \mathtt{v} \mapsto \mathtt{C}(\overline{\mathtt{v}})$ and $\mathit{last}(\mathtt{v}, \rho')$ if $\rho = v\rho' \,.\, \mathtt{v}' \mapsto \mathtt{C}(\overline{\mathtt{v}})$ and $\mathtt{v} \neq \mathtt{v}'$. The function $\mathit{first}(\mathtt{v}, \rho)$ is similar but returns the first binding for $\mathtt{v}$ in the sequence. Finally, $\mathit{findlast}(\mathtt{v}, \mathcal{E})$ finds the last binding for $\mathtt{v}$ in environment $\mathcal{E}$.

There are four computational core reduction rules shown in in Fig. 4. (R-Field) evaluates a field access expression. (R-Assign) evaluates an assignment expression, (R-Invk) evaluates a method invocation expression and (R-New) evaluates an object instantiation expression. Notice that TFJ has a by-value semantics which requires that arguments be fully evaluated before performing method invocations or field access. These rules are complemented by five global reduction rules. (G-Plain) corresponds to a step of computation, (G-Spawn) corresponds to a thread creation, (G-Trans) corresponds to the start of a new transaction, (G-Comm) corresponds to the commit of a transaction, and (G-ThKill) is a reclamation rule for threads in normal form. Most of the rules are straightforward. G-Plain makes use of a *reflect* operation that must propagate the action performed to other threads executing within this transaction; its specification is dependent on the particular transactional semantics adopted. Notice that (G-Comm) requires that, if some thread $\mathtt{t}$ running in transaction $\overline{\mathtt{l}}$ is ready to commit, all other threads executing in that transaction be ready to commit. The auxiliary predicate $\mathit{intranse}(\mathtt{l}, \Gamma)$ given in Fig. 5 returns the set of threads that currently have the transaction label $\overline{\mathtt{l}}$. Note that if there is any thread running in a nested transaction (*e.g.*, has label $\overline{\mathtt{l}}\mathtt{l}'$, for some l'), $\mathit{intranse}(\overline{\mathtt{l}}, \Gamma)$ will return the empty sequence as nested transactions must commit before their parent transaction. The (G-ThKill) rule takes care of removing threads that have terminated, to prevent blocking a transaction (terminated threads are not ready

Evaluation Contexts:

$$E[\bullet] \;\mid\; E[\bullet].\mathtt{f} := \mathtt{e} \;\mid\; \mathtt{e.f} := E[\bullet] \;\mid\; E[\bullet].\mathtt{m}(\overline{\mathtt{e}}) \;\mid\; \mathtt{e.m}(\ldots E[\bullet] \ldots)$$

Congruence:

$$\frac{\mathcal{E}\ \mathtt{e} \longrightarrow \mathcal{E}'\ \mathtt{e}'}{\mathcal{E}\ E[\mathtt{e}] \longrightarrow \mathcal{E}'\ E[\mathtt{e}']}$$

$$\frac{\mathtt{e} = E[\mathtt{spawn}\ \mathtt{e}''] \quad \mathtt{e}' = E[\mathtt{null}]}{\mathtt{e} \Downarrow_{\mathtt{spawn}} \mathtt{e}', \mathtt{e}''}$$

$$\frac{\mathtt{e} = E[\mathtt{onacid}] \quad \mathtt{e}' = E[\mathtt{null}]}{\mathtt{e} \Downarrow_{\mathtt{onacid}} \mathtt{e}'}$$

$$\frac{\mathtt{e} = E[\mathtt{commit}] \quad \mathtt{e}' = E[\mathtt{null}]}{\mathtt{e} \Downarrow_{\mathtt{commit}} \mathtt{e}'}$$

Transaction membership:

$$\frac{}{\mathit{nested}(\mathtt{l}, \langle\,\rangle) = \langle\,\rangle} \qquad \frac{\Gamma = \mathtt{t}, \mathcal{E} \,.\, \Gamma' \quad \mathit{nested}(\mathtt{l}, \Gamma') = \overline{\mathtt{t}} \quad \overline{\mathtt{l}}\mathtt{l}\mathtt{l}'\overline{\mathtt{l}}' = \ell(\mathtt{t}, (\mathtt{t}, \mathcal{E}))}{\mathit{nested}(\mathtt{l}, \Gamma) = \mathtt{t}\overline{\mathtt{t}}} \qquad \frac{\Gamma = \mathtt{t}, \mathcal{E} \,.\, \Gamma' \quad \mathit{nested}(\mathtt{l}, \Gamma') = \overline{\mathtt{t}} \quad \mathtt{l} \notin \ell(\mathtt{t}, (\mathtt{t}, \mathcal{E})) \vee \overline{\mathtt{l}}\mathtt{l} = \ell(\mathtt{t}, (\mathtt{t}, \mathcal{E}))}{\mathit{nested}(\mathtt{l}, \Gamma) = \overline{\mathtt{t}}}$$

$$\frac{}{\mathit{intranse}(\mathtt{l}, \langle\,\rangle) = \langle\,\rangle} \qquad \frac{\Gamma = \mathtt{t}, \mathcal{E} \,.\, \Gamma' \quad \mathit{intranse}(\mathtt{l}, \Gamma') = \overline{\mathtt{t}} \quad \mathtt{l} \in \ell(\mathtt{t}, (\mathtt{t}, \mathcal{E})) \quad \mathit{nested}(\mathtt{l}, \Gamma) = \langle\,\rangle}{\mathit{intranse}(\mathtt{l}, \Gamma) = \mathtt{t}\overline{\mathtt{t}}} \qquad \frac{\Gamma = \mathtt{t}, \mathcal{E}\ \Gamma' \quad \mathit{intranse}(\mathtt{l}, \Gamma') = \overline{\mathtt{t}} \quad \mathtt{l} \notin \ell(\mathtt{t}, (\mathtt{t}, \mathcal{E}))}{\mathit{intranse}(\mathtt{l}, \Gamma) = \overline{\mathtt{t}}}$$

Fig. 5. Auxiliary definitions

to commit). Note also that $\Gamma = \mathtt{t}_0, \mathcal{E} \;.\; \Gamma''$ is used to extract the environment of *one* of the threads in $\overline{\mathtt{t}}$. Since all threads in $\overline{\mathtt{t}}$ execute in the same transaction all of their environments $\overline{\mathcal{E}}$ are identical.

The dynamic semantics leaves open the specification of a number of operations. In particular, the definitions of *read*, *write*, *spawn*, *extend*, *reflect*, *start*, and *join* are left unspecified. A particular incarnation of a transactional semantics must provide a specification for these operations.

3 Versioning Semantics

In Fig. 6 we define an instantiation of TFJ in which transactions implement sequences of object versions. The versioning semantics extends the notion of transaction environments to be an ordered sequence of pairs, each pair consisting of a transaction label and an environment. The intuition is that every transaction operates using a private *log*; these logs are treated as sequences of pairs, bindings a reference to its value. A log thus records effects that occur while executing within the transaction. A given reference may have different binding values in different logs. If $\mathcal{E} = \mathtt{l}_1{:}\rho_1 \;.\; \mathtt{l}_2{:}\rho_2$ then a thread $\mathtt{t}$ executing with respect to this transaction environment is evaluating expressions whose effects are recorded in log ρ_2 and which are part of the dynamic context of an `onacid` command with label $\mathtt{l}_2$. If $\mathtt{l}_2$ successfully commits, bindings in ρ_2 are merged with those in ρ_1. Once $\mathtt{l}_2$ commits, subsequent expressions evaluated by $\mathtt{t}$ occur within the dynamic context of an `onacid` command with label $\mathtt{l}_1$; effects performed by these expressions are recorded in environment ρ_1.

Thus, a transaction environment in a versioning semantics defines a chain of nested transactions: every $\mathtt{l}{:}\rho$ element in $\mathcal{E}$ is related to its predecessor in the sequence defined by $\mathcal{E}$ under an obvious static nesting relationship. A locus of computation can be therefore uniquely denoted by a thread $\mathtt{t}$ and the transaction label sequence $\overline{\mathtt{l}}$ in which $\mathtt{t}$ is executing.

When a new thread is created (cf. *spawn*), the global state is augmented to include the new thread; evaluation of this thread occurs in a transaction environment inherited from its parent. In other words, a spawned thread begins evaluation in the environment of its parent extant at the point where the thread was created.

When a thread enters a new transaction (cf. *start*), a new transaction environment is added to its state. This environment is represented as a pair consisting of a label denoting the transaction, and a log used to hold bindings for objects manipulated within the transaction. Initially, the newly created transaction is bound to an empty log.

The essence of the versioning semantics is captured by the *read*, *write*, and *commit* operations. If a *read* operation on reference $\mathtt{v}$ occurs within transaction $\mathtt{l}$, the last value for $\mathtt{v}$ in the log is returned via the auxiliary procedure *findlast*, and the log associated with `l` is augmented to include this binding. Thus, the first *read* operation for reference `v` within transaction `l` will bind a value for `v` computed by examining the logs of `l`'s enclosing transactions, choosing the binding value found in the one closest to `l`. Subsequent reads of `v` made within `l` will find a binding value within `l`'s log. Thus, this semantics ensures an isolation property on reads: once an object is read within transaction `l`, effects on that object performed within other transactions are not visible until `l` attempts to commit its changes.

The *write* operation is defined similarly. Note that *write* augments the log of the current transaction with two bindings, one binding the reference to its value

$$\frac{\mathcal{E} = \mathcal{E}' \,.\, \mathtt{l}{:}\rho \quad \mathit{findlast}(\mathtt{v}, \mathcal{E}) = \mathtt{C}(\overline{\mathtt{v}}) \qquad \mathcal{E}'' = \mathcal{E}' \,.\, \mathtt{l}{:}(\rho \,.\, \mathtt{v} \mapsto \mathtt{C}(\overline{\mathtt{v}}))}{\mathit{read}(\mathtt{v}, \mathcal{E}) \;=\; \mathtt{C}(\overline{\mathtt{v}}), \mathcal{E}''} \qquad \frac{\mathcal{E} = \mathcal{E}' \,.\, \mathtt{l}{:}\rho \quad \mathit{findlast}(\mathtt{v}, \mathcal{E}) = \mathtt{D}(\overline{\mathtt{u}}) \qquad \mathcal{E}'' = \mathcal{E}' \,.\, \mathtt{l}{:}(\rho \,.\, \mathtt{v} \mapsto \mathtt{D}(\overline{\mathtt{u}}) \,.\, \mathtt{v} \mapsto \mathtt{C}(\overline{\mathtt{v}}))}{\mathit{write}(\mathtt{v} \mapsto \mathtt{C}(\overline{\mathtt{v}}), \mathcal{E}) = \mathcal{E}''}$$

$$\frac{\mathcal{E} = \mathcal{E}' \,.\, \mathtt{l}{:}\rho \qquad \mathcal{E}'' = \mathcal{E}' \,.\, \mathtt{l}{:}(\rho \,.\, \mathtt{x} \mapsto \mathtt{C}(\overline{\mathtt{v}}))}{\mathit{extend}(\mathtt{v} \mapsto \mathtt{C}(\overline{\mathtt{v}}), \mathcal{E}) = \mathcal{E}''}$$

$$\frac{\Gamma = \mathtt{t}, \mathcal{E} \,.\, \Gamma' \quad \Gamma'' = \mathtt{t}', \mathcal{E} \,.\, \Gamma}{\mathit{spawn}(\mathtt{t}, \mathtt{t}', \Gamma) = \Gamma''} \qquad \frac{\Gamma = \mathtt{t}, \mathcal{E} \,.\, \Gamma' \quad \Gamma'' = \mathtt{t}, (\mathcal{E} \,.\, \mathtt{l}{:}\langle\,\rangle) \,.\, \Gamma}{\mathit{start}(\mathtt{l}, \mathtt{t}, \Gamma) = \Gamma''}$$

$$\frac{}{\mathit{reflect}(\mathtt{t}, \mathcal{E}, \langle\,\rangle) = \langle\,\rangle} \qquad \frac{\Gamma = \mathtt{t}', \mathcal{E}' \,.\, \Gamma' \quad \mathit{reflect}(\mathtt{t}, \mathcal{E}, \Gamma') = \Gamma'' \qquad \mathit{copy}(\mathcal{E}, \mathcal{E}') = \mathcal{E}'' \quad \Gamma''' = \mathtt{t}', \mathcal{E}'' \,.\, \Gamma''}{\mathit{reflect}(\mathtt{t}, \mathcal{E}, \Gamma) = \Gamma'''}$$

$$\frac{\mathcal{E} = \mathcal{E}' \,.\, \mathtt{l}{:}\rho \quad \mathit{readset}(\rho, \langle\,\rangle) = \rho' \quad \mathit{writeset}(\rho, \langle\,\rangle) = \rho'' \, \mathit{check}(\rho', \mathcal{E}') \qquad \mathcal{E}' = \mathcal{E}'' \,.\, \mathtt{l}'{:}\rho''' \quad \mathit{reflect}(\mathtt{t}, (\mathcal{E}'' \,.\, \mathtt{l}'{:}\rho'''.\rho''), \Gamma) = \Gamma'}{\mathit{commit}(\mathtt{t}, \mathcal{E}, \Gamma) = \Gamma'}$$

$$\frac{\mathcal{E} = \overline{\mathtt{l}{:}\rho} \,.\, \overline{\mathtt{l}'{:}\rho'} \quad \mathcal{E}' = \overline{\mathtt{l}{:}\rho''} \,.\, \overline{\mathtt{l}''{:}\rho'''}}{\mathit{copy}(\mathcal{E}, \mathcal{E}') = \overline{\mathtt{l}{:}\rho} \,.\, \overline{\mathtt{l}''{:}\rho'''}} \qquad \frac{}{\mathit{check}(\langle\,\rangle, \mathcal{E})} \qquad \frac{\mathit{findlast}(\mathtt{v}, \mathcal{E}) = \mathtt{C}(\overline{\mathtt{v}}) \quad \mathit{check}(\rho, \mathcal{E})}{\mathit{check}(\rho \,.\, \mathtt{v} \mapsto \mathtt{C}(\overline{\mathtt{v}}), \mathcal{E})}$$

Mod sets:

$$\frac{}{\mathit{readset}(\langle\,\rangle, \_) = \langle\,\rangle} \qquad \frac{\rho = \mathtt{u} \mapsto \mathtt{C}(\overline{\mathtt{u}}) \,.\, \rho'' \quad \mathtt{u} \notin \overline{\mathtt{v}} \quad \mathit{readset}(\rho'', \overline{\mathtt{v}}\mathtt{u}) = \rho'}{\mathit{readset}(\rho, \overline{\mathtt{v}}) = \mathtt{u} \mapsto \mathtt{C}(\overline{\mathtt{u}}') \,.\, \rho'}$$

$$\frac{\rho = \mathtt{u} \mapsto \mathtt{C}(\overline{\mathtt{u}}) \,.\, \rho'' \quad \mathtt{u} \in \overline{\mathtt{v}} \quad \mathit{readset}(\rho'', \overline{\mathtt{v}}) = \rho'}{\mathit{readset}(\rho, \overline{\mathtt{v}}) = \rho'}$$

$$\frac{}{\mathit{writeset}(\langle\,\rangle, \_) = \langle\,\rangle} \qquad \frac{\rho = \mathtt{v} \mapsto \mathtt{C}(\overline{\mathtt{v}}) \,.\, \rho'' \quad \mathit{writeset}(\rho'', \rho') = \rho''' \qquad \mathtt{v} \mapsto \mathtt{C}(\overline{\mathtt{v}}) \neq \mathit{first}(\mathtt{v}, \rho')}{\mathit{writeset}(\rho, \rho') = \mathtt{u} \mapsto \mathtt{D}(\overline{\mathtt{u}}) \,.\, \rho'''}$$

Fig. 6. Versioning semantics

prior to the assignment, and the other reflecting the effect of the assignment. The former binding is needed to guarantee transactional consistency. Consider a write to a reference `v` in transaction `l` which has not yet been read or written in `l`. The effects of this write can be made visible when `l` attempts to commit only if no other transaction has committed modifications to `v` in the interim between the time where the write occurred, and `l` attempts to commit. If this invariant were violated, the desired serialization semantics on transaction would fail to hold. The *extend* operation inserts a new binding in the current transaction's log; since the reference being bound is fresh, there is no existing binding in the parent transaction against which to check consistency upon commit.

The *commit* operation is responsible for committing a transaction. In our versioning semantics, a commit results in bindings for objects written within a transaction's log to be propagated to its parent. In order for a commit of transaction `l` to succeed, it must be the case that the binding value of every reference read or written in `l` must be the same as its current value in `l`'s parent transaction. Satisfaction of this condition implies the absence of a data race between `l` and its parent or siblings. The *reflect* operation defined in *commit* makes visible the effects of `l` in all threads executing in `l`'s parent transaction; when used in a transaction-local action, it propagates the effects of the action to other threads executing within this same transaction.

The versioning semantics defined here is akin to an optimistic concurrency protocol in which the validity of reads and writes of references performed within a transaction `l` is determined by the absence of modifications to those references in transactions which commit between the time the first read or write of the reference takes place in `l` and the time `l` commits. For example, consider transaction $\mathtt{l}_1$ that commits $\mathtt{v}_1$, transaction $\mathtt{l}_2$ that commits $\mathtt{v}_2$ and transaction `l` that accesses both $\mathtt{v}_1$ and $\mathtt{v}_2$; a valid serialization of these transactions would commit $\mathtt{l}_1$ prior to the first access of $\mathtt{v}_1$ in $\mathtt{l}_2$, and would commit $\mathtt{l}_2$ prior to the first access of $\mathtt{v}_2$ in `l`. Provided $\mathtt{l}_2$ does not modify $\mathtt{v}_1$, no atomicity or consistency invariants on these transactions would be violated.

4 Strict Two-Phase Locking

With slight alteration, the versioning semantics can be modified to support a two-phase locking protocol. The semantics presented below is faithful to a two-phase locking protocol in which locks are first acquired on objects before the objects can be accessed, and released only when commit actions occur. The modifications necessary are shown in Fig. 7. The primary change is in the definition of *reflect*. In the versioning semantics, and in the global reduction rules, the *reflect* operator is used to propagate changes performed in one thread to all other threads executing within the same transaction; it is also used in the definition of commit to propagate updates to a parent transaction to all threads that execute within it.

To support two-phase locking, we exploit this functionality to allow transaction environments to reflect object ownership. We define a unique transaction environment $\mathcal{E}_L$ containing a unique log ρ_L; $\rho_L(\mathtt{v})$ maps $\mathtt{v}$ to the transaction label sequence which identifies the transaction that currently has exclusive access to $\mathtt{v}$. If $\overline{\mathtt{l}} = \mathtt{l}_1.\mathtt{l}_2 \ldots \mathtt{l}_\mathtt{n}$ is such a sequence, and a thread $\mathtt{t}$ executing within $\mathtt{l}_\mathtt{n}$ attempts to read or write reference $\mathtt{v}$, it must first acquire $\mathtt{v}$'s lock. A lock is acquired using *setlock*. A lock of an object can be acquired if (a) the transaction in which the action occurs is a child of the current owner, or (b) the lock is currently owned by the child, and needs to be propagated to the parent. The first condition arises for non-commit actions; in this case the transaction attempting to acquire the lock ($\mathtt{l}$) is the same as the transaction in which the *setlock* is executed. The second condition arises on commit actions; in this case $\mathtt{l}$ is the transaction of the parent to whom ownership of all locks owned by the child $\mathtt{l}_\mathtt{t}$ must be transferred. Thus, locks are reset on a commit: when a commit occurs, lock ownership is changed from $\mathtt{l}$ to $\mathtt{l}$'s parent.

We distinguish between read and write operations and commit operations in the definition of *reflect* and *setlock* by observing that, in the former case, the transaction label sequence of the transaction performing the reflect is the same as the label sequence of the transaction in which the locks are to be acquired (i.e., $\ell(\mathcal{E}) = \ell(\mathcal{E}_t)$); in the latter case, the *reflect* operation is invoked by *commit*, and thus $\mathcal{E}$ is the parent transaction of $\mathcal{E}_t$, the transaction being committed. Recall that the definition of *commit* in Fig. 6 updates the transaction environment by propagating changes performed by the committing transaction to its parent.

5 Soundness

Proving the soundness of a particular transactional facility requires relating it to desired serialization characteristics that dictate a transaction's ACID properties. For any abort-free program trace there must be a corresponding trace in which the transactions executed serially, *i.e.* all concurrent transactions execute atomically wrt one another. The key idea is that we should be able to

$$\frac{\begin{array}{c}\mathcal{E}_L = \mathtt{l}{:}\rho_L \qquad \rho = \mathtt{v} \mapsto \mathtt{C}(\overline{\mathtt{u}}) \;.\; \rho' \qquad \mathit{findlast}(\mathtt{v}, \mathcal{E}_L) = \mathtt{Lock}(\overline{\mathtt{l}}') \\ (\overline{\mathtt{l}}' \lhd \overline{\mathtt{l}} \wedge \;\overline{\mathtt{l}} = \overline{\mathtt{l}}_t) \vee (\overline{\mathtt{l}}_t = \overline{\mathtt{l}} \;.\; \mathtt{l}) \qquad \mathcal{E}'_L = \mathit{setlock}(\rho', \overline{\mathtt{l}}, \mathcal{E}_L) \\ \mathcal{E}'_L = \mathtt{l}{:}\rho'_L \qquad \rho''_L = \mathtt{v} \mapsto \mathtt{Lock}(\overline{\mathtt{l}}) \;.\; \rho'_L \qquad \mathcal{E}''_L = \mathtt{l}{:}\rho''_L\end{array}}{\mathit{setlock}(\rho, \overline{\mathtt{l}}, \overline{\mathtt{l}}_t, \mathcal{E}_L) = \mathcal{E}''_L}$$

$$\frac{\begin{array}{c}\Gamma/_{\mathtt{t_L},\mathcal{E}_\mathtt{L}} = \mathtt{t}, \mathcal{E}_t \;.\; \mathtt{t}', \mathcal{E}' \;.\; \Gamma' \qquad \mathit{reflect}(\mathtt{t}, \mathcal{E}, \Gamma') = \Gamma'' \\ \Gamma''' = \mathtt{t}', \mathcal{E}'' \;.\; \Gamma'' \qquad \mathcal{E} = \mathcal{E}''' \;.\; \overline{\mathtt{l}}{:}\rho \quad \mathit{readset}(\rho, \langle\,\rangle) = \rho' \\ \mathit{setlock}(\rho', \ell(\mathcal{E}), \ell(\mathcal{E}_t), \mathcal{E}_L) = \mathcal{E}'_L \qquad \mathit{copy}(\mathcal{E}, \mathcal{E}') = \mathcal{E}''\end{array}}{\mathit{reflect}(\mathtt{t}, \mathcal{E}, \Gamma) = \Gamma'''}$$

Fig. 7. Lock-based commitment semantics

reorder any abort-free sequence of reduction steps into a sequence that yields the same final state and in which reduction steps taken on behalf of different parallel transactions are not interleaved. We proceed to formalize this intuitive definition.

The height of an environment $\mathcal{E} = \mathtt{l}_0{:}\rho_0 \dots \mathtt{l}_n{:}\rho_n$, written $|\mathcal{E}|$, is n. For a state Γ, $max(\Gamma)$ returns a thread environment $\mathtt{t}, \mathcal{E}$ such that $\mathcal{E}$ is the environment with the largest height $|\mathcal{E}|$ in Γ. Given a transition $P\,\Gamma \overset{\alpha}{\Longrightarrow}_{\mathtt{t}} P'\,\Gamma'$, we say that the corresponding action, written $\mathcal{A}$ is $(\alpha, \mathtt{t}, \ell(\mathtt{t}, \mathcal{E}))$.

Definition 1 (Well-defined). *Let $\Gamma = (t, \mathcal{E})\ .\ \Gamma'$. We say that environment Γ is well-defined if Γ' is also well-defined and for $\mathcal{E} = \mathtt{l}_1{:}\rho_1 \dots \mathtt{l}_n{:}\rho_n$, we have* first$(\rho_j, \mathtt{v}) =$ last$(\rho_{j-1}, \mathtt{v})$ *if $2 \leq j \leq n$, and $\mathtt{v} \in Dom(\rho_{j-1}) \cap Dom(\rho_j)$.*

To define soundness properties, we introduce the notion of control and data dependencies. A dependency defines a relation on actions which can be used to impose structure of transition sequences. In other words, a well-defined transition sequence will be one in which action dependencies are not violated, and thus define safe serial orderings.

Definition 2 (Control Dependency). *Define a preorder $\overset{c}{\leadsto}$ on actions such that $\mathcal{A}_1 \overset{c}{\leadsto} \mathcal{A}_2$ (read $\mathcal{A}_1$ is control-dependent on $\mathcal{A}_2$) if the following holds:*
1. $\mathcal{A}_1 = (\alpha, \mathtt{t}, \overline{\mathtt{l}})$ and $\mathcal{A}_2 = (sp\,\mathtt{t}, \mathtt{t}', \overline{\mathtt{l}})$.
2. $\mathcal{A}_1 = (co, \mathtt{t}, \overline{\mathtt{l}})$ and $\mathcal{A}_2 = (\alpha, \mathtt{t}', \overline{\mathtt{l}}')$ where $\alpha \in \{rd, wr, xt\}$ and $\overline{\mathtt{l}}' \lhd \overline{\mathtt{l}}$.
3. $\mathcal{A}_1 = (\alpha, \mathtt{t}, \overline{\mathtt{l}})$ and $\mathcal{A}_2 = (ac, \mathtt{t}', \overline{\mathtt{l}}')$ where $\overline{\mathtt{l}}' \lhd \overline{\mathtt{l}}$.

Definition 3 (Data Dependency). *Define a preorder $\overset{d}{\leadsto}$ on actions such that $\mathcal{A}_1 \overset{d}{\leadsto} \mathcal{A}_2$ (read $\mathcal{A}_1$ is data-dependent on $\mathcal{A}_2$) the if $\mathcal{A}_1$ is either $(rd\,\mathtt{v}, \mathtt{t}, \overline{\mathtt{l}})$, $(wr\,\mathtt{v}\mathtt{v}', \mathtt{t}, \overline{\mathtt{l}})$ or $(wr\,\mathtt{v}'\mathtt{v}, \mathtt{t}, \overline{\mathtt{l}})$, and $\mathcal{A}_2$ is either $(wr\,\mathtt{v}\mathtt{v}'', \mathtt{t}', \overline{\mathtt{l}}')$ or $(xt\,\mathtt{v}, \mathtt{t}', \overline{\mathtt{l}}')$, with $\overline{\mathtt{l}'} \lhd \overline{\mathtt{l}}$.*

The key property for our soundness result is the permutation lemma which describes the conditions under which two reduction steps can be permuted. Let $\mathcal{A}$ and $\mathcal{A}'$ be a pair of actions which are not related under a control or data dependency. We write $\mathcal{A} \overset{\overline{d}}{\leadsto} \mathcal{A}'$ and $\mathcal{A} \overset{\overline{c}}{\leadsto} \mathcal{A}'$ to mean action $\mathcal{A}$ has, respectively, no c-dependence or d-dependence on $\mathcal{A}'$.

Definition 4 (Independence). *Actions $\mathcal{A}$ and $\mathcal{A}'$ are independent if $\mathcal{A} \overset{\overline{c}}{\leadsto} \mathcal{A}'$ and $\mathcal{A} \overset{\overline{d}}{\leadsto} \mathcal{A}'$.*

Lemma 1 (Permute). *Assume that Γ and Γ'' are well-defined, and let R be the two-step sequence of reductions $P\,\Gamma \overset{\alpha}{\Longrightarrow}_{\mathtt{t}} P_0\,\Gamma_0 \overset{\alpha'}{\Longrightarrow}_{\mathtt{t}'} P'\,\Gamma'$. If $\mathcal{A}$ and $\mathcal{A}'$ are independent then there exists a two-step sequence R' such that R' is $P\,\Gamma \overset{\alpha'}{\Longrightarrow}_{\mathtt{t}'} P_1\,\Gamma_1 \overset{\alpha}{\Longrightarrow}_{\mathtt{t}} P'\,\Gamma'$.*

Definition 5 (Program Trace). *Let R be the sequence of reductions $P_0\,\Gamma_0 \overset{\alpha_0}{\Longrightarrow}_{t_0} \ldots P_n\,\Gamma_n \overset{\alpha_n}{\Longrightarrow}_{t_n} P_{n+1}\,\Gamma_{n+1}$. The trace of the reduction sequence R, written $tr(R)$, is $(\alpha_0, \mathtt{t}_0, \overline{\mathtt{l}}_0) \ldots (\alpha_n, \mathtt{t}_n, \overline{\mathtt{l}}_0)$ assuming that $\overline{\mathtt{l}}_i = \ell(\mathtt{t}_i, \Gamma_i)$ for $0 \leq i \leq n$.*

A program trace is *serial* if for all pairs of reduction steps with the same transaction label ($\overline{\mathtt{l}}$), all reductions occurring between the two steps are taken on behalf of that very transaction or nested transactions ($\overline{\mathtt{l}} \lhd \overline{\mathtt{l}}'$).

Definition 6 (Serial Trace). *A program trace, $tr(R) = (\alpha_0, \mathtt{t}_0, \overline{\mathtt{l}}_0) \ldots (\alpha_n, \mathtt{t}_n, \overline{\mathtt{l}}_n)$ is serial iff $\forall\, i, j, k$ such that $0 \leq i \leq j \leq k \leq n$ and $\overline{\mathtt{l}}_i = \overline{\mathtt{l}}_k$ we have $\overline{\mathtt{l}}_i \lhd \overline{\mathtt{l}}_j$.*

We can now formulate the soundness theorem which states that any sequence of reductions which ends in a good state can be reordered so that its program trace is serial.

Theorem 1 (Soundness). *Let R be a sequence of reductions $P_0\,\Gamma_0 \overset{\alpha_0}{\Longrightarrow}_{t_0} \ldots P_n\,\Gamma_n \overset{\alpha_n}{\Longrightarrow}_{t_n} P_{n+1}\,\Gamma_{n+1}$. If Γ_{n+1} is well-defined, then there exists a sequence R' such that R' is $P_0\,\Gamma_0 \overset{\alpha_0'}{\Longrightarrow}_{t_0'} \ldots P_n'\,\Gamma_n' \overset{\alpha_n'}{\Longrightarrow}_{t_n'} P_{n+1}\,\Gamma_{n+1}$ and $tr(R')$ is serial.*

6 Related Work

The association of transactions with programming control structures has provenance in systems such as Argus [17,15,18], Camelot [10] Avalon/C++ [9] and Venari/ML [13], and has also been studied for variants of Java, notably by Garthwaite [11] and Daynes [6,7,8]. There is a large body of work that explores the formal specification of various flavors of transactions [16,5,12]. However, these efforts do not explore the semantics of transactions when integrated into a high-level programming language. Most closely related to our goals is the work of Black *et. al.* [1] and Choithia and Duggan [4]. Choithia and Duggan present an extension of the pi-calculus that supports various abstractions for distributed transactions and optimistic concurrency. Their work is related to other efforts [3, 2] that encode transaction-style semantics into the pi-calculus and its variants. Our work is distinguished from these efforts in that it provides a simple operational characterization and proof of correctness of transactions that can be used to explore different trade-offs when designing a transaction facility for incorporation into a language.

7 Conclusions

This paper presented a semantic framework for specifying nested and mulithreaded transactions. The TFJ calculus is an object calculus which supports nested and multi-threaded transactions and enjoys a semantics parameterized by the definition of the transactional facility. We have proven a general soundness theorem that relates the semantics of TFJ to a serializability property, and have defined two instantiations: a versioning-based optimistic model, and a pessimistic two-phase locking protocol. In future work we plan to address typing issues as well as static analysis techniques for optimized implementations of transactional languages. Furthermore we plan to investigate higher-order transactions.

Acknowledgments. This work is supported by the National Science Foundation under grants Nos. IIS-9988637, CCR-0085792, 0093282, and 0209083, and by gifts from Sun Microsystems and IBM. The authors thank Krzysztof Palacz for his input.

References

1. Andrew Black, Vincent Cremet, Rachid Guerraoui, and Martin Odersky. An Equational Theory for Transactions. Technical Report CSE 03-007, Department of Computer Science, OGI School of Science and Engineering, 2003.
2. R. Bruni, C. Laneve, and U. Montanari. Orchestrating Transactions in the Join Calculus. In *13th International Conference on Concurrency Theory*, 2002.
3. N. Busi, R. Gorrieri, and G. Zavattaro. On the Serializability of Transactions in JavaSpaces. In *ConCoord 2001, International Workshop on Concurrency and Coordination*, 2001.
4. Tom Chothia and Dominic Duggan. Abstractions for Fault-Tolerant Computing. Technical Report 2003-3, Department of Computer Science, Stevens Institute of Technology, 2003.
5. Panos Chrysanthis and Krithi Ramamritham. Synthesis of Extended Transaction Models Using ACTA. *ACM Transactions on Database Systems*, 19(3):450–491, 1994.
6. Laurent Daynès. Implementation of automated fine-granularity locking in a persistent programming language. *Software—Practice and Experience*, 30(4):325–361, April 2000.
7. Laurent Daynès and Grzegorz Czajkowski. High-performance, space-efficient, automated object locking. In *Proceedings of the International Conference on Data Engineering*, pages 163–172. IEEE Computer Society, 2001.
8. Laurent Daynès and Grzegorz Czajkowski. Lightweight flexible isolation for language-based extensible systems. In *Proceedings of the International Conference on Very Large Data Bases*, 2002.
9. D. D. Detlefs, M. P. Herlihy, and J. M. Wing. Inheritance of synchronization and recovery in Avalon/C++. *IEEE Computer*, 21(12):57–69, December 1988.
10. Jeffrey L. Eppinger, Lily B. Mummert, and Alfred Z. Spector, editors. *Camelot and Avalon: A Distributed Transaction Facility*. Morgan Kaufmann, 1991.
11. Alex Garthwaite and Scott Nettles. Transactions for Java. In Malcolm P. Atkinson and Mick J. Jordan, editors, *Proceedings of the First International Workshop on Persistence and Java*, pages 6–14. Sun Microsystems Laboratories Technical Report 96-58, November 1996.
12. Jim Gray and Andreas Reuter. *Transaction Processing*. Morgan-Kaufmann, 1993.

13. Nicholas Haines, Darrell Kindred, J. Gregory Morrisett, Scott M. Nettles, and Jeannette M. Wing. Composing first-class transactions. *ACM Transactions on Programming Languages and Systems*, 16(6):1719–1736, November 1994.
14. Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, May 2001.
15. B. Liskov and R. Scheifler. Guardians and actions: Linguistic support for robust distributed programs. *ACM Transactions on Programming Languages and Systems*, 5(3):381–404, July 1983.
16. Nancy Lynch, Michael Merritt, William Weihl, and Alan Fekete. *Atomic Transactions*. Morgan-Kaufmann, 1994.
17. J. Eliot B. Moss. Nested transactions: An approach to reliable distributed computing. In *Proceedings of the Second Symposium on Reliability in Distributed Software and Database Systems*, pages 33–39. IEEE Computer Society, 1982.
18. J. Eliot B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. MIT Press, Cambridge, Massachusetts, 1985.