Microsoft Tech Report MSR-TR-2010-94

Semantics of Concurrent Revisions

Sebastian Burckhardt
Microsoft Research
sburckha@microsoft.com

Daan Leijen
Microsoft Research
daan@microsoft.com

Abstract

Enabling applications to execute various tasks in parallel is difficult if those tasks exhibit read and write conflicts. We recently developed a programming model based on *concurrent revisions* that addresses this challenge in a novel way: each forked task gets a conceptual copy of all the shared state, and state changes are integrated only when tasks are joined, at which time write-write conflicts are deterministically resolved.

In this paper, we study the precise semantics of this model, in particular its guarantees for determinacy and consistency. First, we introduce a revision calculus that concisely captures the programming model. Despite allowing concurrent execution and locally nondeterministic scheduling, we prove that the calculus is confluent and guarantees determinacy. We show that the consistency guarantees of our calculus are a logical extension of snapshot isolation with support for conflict resolution and nesting. Moreover, we discuss how custom merge functions can provide stronger guarantees for particular data types that are tailored to the needs of the application.

Finally, we show we can visualize the nonlinear history of state in our computations using *revision diagrams* that clarify the synchronization between tasks and allow local reasoning about state updates.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

General Terms term1, term2

Keywords keyword1, keyword2

1. Introduction

With the recent broad availability of shared-memory multiprocessors, many more application developers now have a strong motivation to tap into the potential performance benefits of parallel execution. Exploiting parallel hardware can be relatively easy if the application performs computations for which parallel algorithms are well known or straightforward to develop (such as for scientific problems or multimedia applications). However, traditional parallelization strategies often do not satisfactorily address how to execute different application tasks that access shared data in parallel.


```
x = 0 ;
y = 0 ;

r = rfork { x = 1 ; }
y = x ;
rjoin r ;

print x, y ;
```

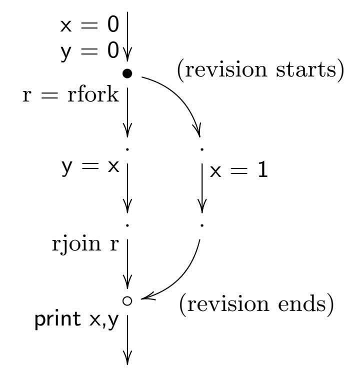


Figure 1. An example of a revision diagram (on the right) representing the execution of a program (on the left). The effect of the write $x = 1$ is confined to its revision until that revision is joined. Thus the print statement prints (1,0).

For example, consider an office application that needs to perform five different tasks: (1) save a snapshot of the document to disk, (2) react to keyboard input by the user who is editing the document, (3) perform a spellcheck of the document, (4) render the document on the screen, and (5) exchange document updates with collaborating remote users.

Executing such tasks in parallel is not simple, because all of them potentially access the same data (such as the document) at the same time. Avoiding, negotiating, or resolving conflicts can be quite challenging with traditional synchronization models. In fact, many programmers do not even attempt parallelization of such tasks as they are deterred by the complexity of performing explicit, manual synchronization (such as by using locks and critical sections) or replication (such as by creating temporary copies).

We recently proposed a new programming model, *concurrent revisions*, which simplifies parallelization of such tasks by (conceptually) copying shared state automatically on a fork. Tasks execute in complete isolation because each has its own copy of the shared data (e.g. the document in the above example), somewhat analogous to source control systems that allow multiple programmers to work on the same code at the same time by creating local copies of files, and checking changed files back into the repository.

For example, consider the code in Fig. 1 which illustrates the basic concept of *forking and joining revisions* and how to visualize executions using *revision diagrams*. The program on the left forks a concurrent revision, obtaining a handle r which it later joins. The forked revision executes the assignment $x = 1$, but the effect of this assignment is confined to that revision until it is joined, at which point all of its changes are applied to the joining revision. The diagram shows how the state is forked and joined (each vertex represents a state, and curved arrows represent fork and join), as well as how the state is locally updated by revisions (vertical arrows represent steps by revisions). Note that because revisions are isolated, data can flow only along edges in the diagram. Moreover, because the program specifies where to join revisions and does not depend on scheduling and timing, the execution is determinate.

Our previous work [8] has demonstrated that this concurrent revision model can be implemented efficiently enough to achieve satisfactory parallelization speedups without requiring the introduction of locks and critical sections, or of manual state replication. However, our previous work has only partially addressed important questions about the semantics, in particular questions relating to determinacy and consistency guarantees. The purpose of our work presented in this paper is to address these questions rigorously and provide precise answers. We make the following contributions:

1. We give a minimal calculus describing the concurrent revision model. Because the calculus is small, it is well suited as a semantic reference and as an experimental tool to study various extensions or implementations. In fact, it was inspired (and is very similar to) the AME calculus [24] which served a similar purpose in the context of transactional memory.
2. Even though the calculus is intrinsically concurrent, we prove that it guarantees determinacy.
3. We give a comprehensive discussion of consistency guarantees and state merging. We show that in the absence of write-conflicts and nesting, revisions are analogous to transactions with snapshot isolation. We also show how the introduction of custom *merge* functions into the calculus can allow the programmer to achieve stronger consistency guarantees tailored to the needs of the application.
4. We formalize the notion of a *revision diagram.* These diagrams capture the revision history of an execution, by showing the order and nesting of forks and joins. Moreover, they illustrate data flow, since information can propagate only along edges. We prove that revision diagrams are semilattices, which means that we can always find a greatest common ancestor when merging states.

Overall, our work shows that the revision model preserves some of the best properties of sequential programs (deterministic execution, local reasoning about state updates) without forcing programmers to manually isolate parallel tasks, and without restricting parallel executions to be fully equivalent to a sequential execution. Rather, parallelism is expressed directly and explicitly, and always exploitable even if the tasks exhibit conflicts.

2. Discussion

We start with a high-level informal discussion of various aspects of the revision model, such as determinacy, nesting of revisions, handling of write-write conflicts, and revision diagrams. Moreover, we compare revisions to related work on transactional memory and determinacy.

2.1 Revisions vs. Interleaved Tasks

In our model, revisions are the basic unit of concurrency. They function much like asynchronous tasks that are forked and joined, and they may themselves fork and join other tasks. We chose the term 'revision' to emphasize the semantic similarity to branches in source control systems where programmers work with a local snapshot of the shared source code.

In particular, on every revisional fork (rfork), the system conceptually copies the entire state and each branch works on its own local copy. Every revision is completely isolated from the others and there is no possibility of communication through shared state. Any updates in a revision only become re-integrated once the revision is joined. Since there is no possibility of stateful interleavings with other threads, intra-revision reasoning (that is, reasoning about the behavior of code executing within a revision) is sequential.

The revision model is a significant departure from memory models that interleave tasks at the level of individual instructions, such as sequential consistency [21]. Moreover, this difference is not simply a matter of the interleaving granularity. Transactional memory, for example, interleaves tasks at the granularity of atomic blocks [15, 22]. However, coarser interleaving does not in itself guarantee determinacy of executions, as the relative order of the atomic blocks is not specified by the program. Thus, whether we use sequential consistency or transactional memory, the interleaving chosen during an execution depends on nondeterministic arbitration which can vary between executions. In contrast, with our concurrent revision model, the precise structure of forks and joins is completely determined by the program and independent of runtime scheduling decisions.

We illustrate this difference in Figure 2 where we compare the results of a program for these three models. The program forks a concurrent branch where each branch increments a variable x or y respectively depending on the value of the other variable (y and x respectively). Under sequential consistency, there are many interleavings possible and there are three distinct possibilities for the values of x and y. In the second program, we use transactional memory to limit the possible interleavings by executing each branch atomically. This effectively serializes the execution and we see either $x = 0 \wedge y = 1$ or $x = 1 \wedge y = 0$ depending on how the branches are scheduled. Using revisions, the outcome is always determinate: both branches get their own local (conceptual) copy of the state, and both branches will increment the variables ending in $x = 1 \wedge y = 1$.

2.2 Local Reasoning vs. Serializability

As Figure 2 shows, we can truly reason about each branch locally without considering any interleavings. However, note also that there is no equivalent sequential execution for this example. The lack of equivalence to some sequential execution is no accident: requiring such equivalence fundamentally limits the concurrency that can be practically exploited if tasks exhibit conflicts. For the kind of applications we have in mind, conflicts may be quite frequent.

With revisions, conflicts never destroy the available parallelism and never cause rollbacks. These choices provide substantial practical benefits over the use of rollbacks in optimistic transactional memory, which does not fare well in the presence of frequent conflicts, and which cannot be easily combined with I/O without additional restrictions [33].

Comfortably reasoning about application behavior in the absence of serializability requires understanding and con-

(sequential consistency)

```
x = 0 ; y = 0 ;
t = fork { if (x = 0) y++ ; }
if (y = 0) x++ ;
join t ;

assert( (x = 0 ∧ y = 1) ∨
        (x = 1 ∧ y = 0) ∨
        (x = 1 ∧ y = 1) ) ;
```

(transactional memory)

```
x = 0 ; y = 0 ;
t = fork { atomic { if (x = 0) y++ ; } }
atomic { if (y = 0) x++ ; }
join t ;

assert( (x = 0 ∧ y = 1) ∨
        (x = 1 ∧ y = 0) ) ;
;
```

(concurrent revisions)

```
x = 0 ; y = 0 ;
r = rfork { if (x = 0) y++ ; }
if (y = 0) x++ ;
rjoin r ;

assert( x = 1 ∧ y = 1 ) ;
;
;
```

Figure 2. Outcomes under different programming models.

ceptualizing a nonlinear history of state. We achieve this by introducing revision diagrams that directly visualize how the global state can be forked, updated, and joined (Fig. 1, Fig. 3). Revisions correspond to vertical chains in the diagram, and are connected by curved arrows that represent the forks and joins. We sometimes label the revisions with the actions they perform. Such diagrams visualize clearly how information may flow (it follows the edges) and how effects become visible upon the join. In Section 5 we show that the diagrams have a formal and well-defined meaning with relation to the calculus.

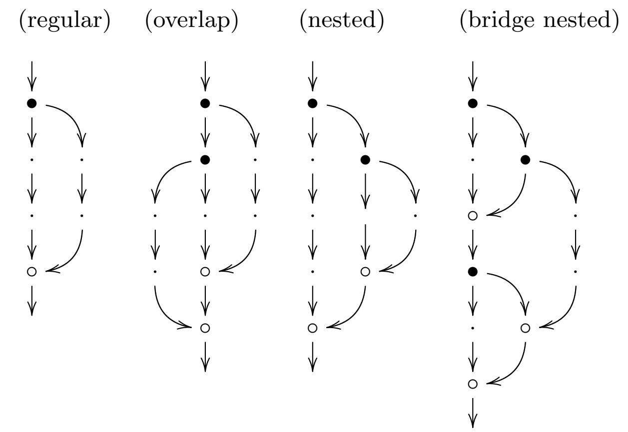


Figure 3. Some examples of revision diagrams

2.3 State Merging

When joining a revision, two copies of the state need to be merged together, which naturally raises two questions:

1. Can we always find a common ancestor state to help us determine if either side has made changes, and what those changes are?
2. If both sides have made changes, how do we resolve such write-write conflicts? (Note that there are no read-write or write-read conflicts between revisions.)

We answer the first question by showing how our calculus keeps track of the ancestor state (Section 3), and by showing that revision diagrams are semilattices and the ancestor state is in fact the greatest common ancestor (Section 5).

We address the second question by discussing several sensible merge policies. A key insight that makes state merging practical and convenient is that we need not define merge functions or policies globally, but can do so separately for each variable. In fact, we used this insight in previous work to parallelize a game application [8] by declaring the policy for each variable using special *isolation types*. Such isolation types allow the user to convey deep semantic knowledge that helps to exploit the available parallelism even if there are numerous conflicts.

In this paper, we consider a number of different merge policies. Note that these happen at the granularity of individual memory locations, not on the global state.

- (Join overwrites). This policy is the default in our basic calculus (Section 3). On a write-write conflict, the value of the joined revision overwrites the value of the joining revision.
- (Custom merge function). We can use a user-defined merge function to resolve conflicts deterministically (Section 4.1).
- (Give up and report). We can refuse to merge write-write conflicts and report the failure to the user, who can take some appropriate action. (Section 4.3).

What we found a bit surprising is that the (Join overwrites)-policy is very useful in practice even though it appears to 'lose state'. This is because it lets us precisely control which revisions should take precedence over others by ordering the joins accordingly. For instance, if writes by revision B should take priority over writes by revision A, we can simply join B after joining A. The (Custom merge function)-policy was useful very specifically for implementing collections, which are often updated in a commutative way by concurrent revisions. We did not have any use for the (Give Up and Report)-policy in the game.

Isolation types are also sensible from a software engineering perspective: in a large application an architect can annotate the shared data structures with their merge policy, while the code that uses such data types stays the same: in particular, programmers have no need to use atomic regions or locks when accessing such data types and can reason about it without considering interleaved executions.

2.4 Nesting of Revisions

Nesting of revisions is a natural consequence of the fact that revisions can themselves fork and join other revisions. We show a progression of nesting in Fig. 3. The (regular) and (overlap) diagrams do not nest revisions beyond a depth of 1 (that is, only the main revision is forking and joining revisions). The (nested) diagram shows simple nesting, where a revision forks a child of depth 2 and then joins it (before being joined itself). The (bridge nested) diagram shows that child revisions can "survive" their parents (i.e. be joined later), and that revisions can be joined by a different revision than where they were forked.

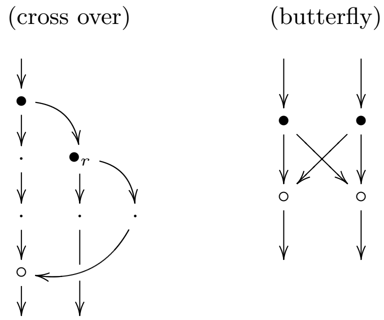


Figure 4. Examples of impossible revision diagrams: the left one is not possible since the main branch cannot join on the outer revision as the (fresh) outer revision handle r cannot be part of its state. The right diagram cannot be constructed for similar reasons, in particular, all revision diagrams are semi-lattices (Theorem 10).

However, not all diagrams are possible, because revision handles must flow along edges. See Fig. 4 for examples of impossible revision diagrams. We prove some structural properties of revision diagrams in Section 5, in particular that revision diagrams are semi-lattices (Theorem 10).

Note that the structure of revision diagrams is entirely dynamic, not lexical. In particular, once a revision is forked, its handle can be stored in arbitrary data structures and be joined at an arbitrary later point of time. In some sense, revisions behave like futures whose side effects are delayed, and take effect atomically at the moment when the future is forced.

Although we present a fully dynamic model, it is of course possible to design a language that statically restricts the use of joins, to make stronger scheduling guarantees (as done in Cilk++ [14, 29]) or to simplify the most common usage patterns and to eliminate common user mistakes (as done in X10 [23]). In fact, many models (including an earlier version of our calculus) use a restricted "fork-join" parallelism [5, 7]. Whether such restrictions are necessary or beneficial is beyond the scope of this paper. For now, we are content with stating that it is relatively easy to add them if desired, while it would be difficult to remove them from a calculus that depends on restrictive assumptions.

2.5 Related Work

Just as we do with revisions, proponents of transactions have long recognized that providing strong guarantees such as serializability [27] or linearizability [17] can be overly conservative for some applications, and have proposed alternate guarantees such as multi-version concurrency control [26] or snapshot isolation (SI) [3, 11, 30]. In fact, revisions can be understood as a natural generalization of snapshot isolation, extended to handle resolution of write-write conflicts following some policy (as discussed in Section 2.3), and to support nesting (as discussed in Section 2.4). We examine the relationship to snapshot isolation more formally in Section 4.3.

There has been much prior work on programming models for concurrency[1, 2, 6, 12, 25, 31]. Recently, many researchers have proposed programming models for deterministic concurrency [5, 7, 28, 32], creating renewed interest in an old problem previously known as determinacy [10]. All of these models differ semantically from revisions, and are quite a bit more restrictive. As they guarantee that the execution is equivalent to some sequential execution, they cannot easily resolve all conflicts on commit (like revisions do). Thus, they must restrict tasks from producing such conflicts either statically (by type system) or dynamically (pessimistic with blocking, or optimistic with abort and retry).

To the best of our knowledge, our combination of snapshot isolation and deterministic conflict resolution, as first presented in [8], is a novel way to simplify the parallelization of tasks that exhibit conflicts.

Isolation types are similar to Cilk++ hyperobjects [13]: both use type declarations by the programmer to change the semantics of shared variables. Cilk++ hyperobjects may split, hold, and reduce values. Although these primitives can (if properly used) achieve an effect similar to revisions, they do not provide a similarly seamless semantics. In particular, the determinacy guarantees are fragile, i.e. do not hold for all programs. For instance, the following program may finish with either $x == 2$ or $x == 1$:

```
reducer_opadd⟨int⟩ x = 0;
cilk_spawn { x++ }
if (x= 0) x++;
cilk_sync
```

Isolation types are also similar to the idea of transactional boosting, coarse-grained transactions, and semantic commutativity [16, 19, 20], which eliminate false conflicts by raising the abstraction level. Isolation types go farther though: for example, the type versioned⟨T⟩ does not just avoid false conflicts, but resolves true conflicts deterministically (in a not necessarily serializable way).

3. Revision Calculus

For reference and to remove potential ambiguities, we now present a formal calculus for revisions. It is based on a similar calculus introduced by prior work on AME (automatic mutual exclusion) [24].

Notations. To present the formal syntax and semantics succinctly, we use some standard and nonstandard notations for partial functions. For sets A, B, we write $A \rightharpoonup B$ for the set of partial functions from A to B. For $f, g \in A \rightharpoonup B$, $a \in A$, $b \in B$, and $A' \subset A$, we adopt the following notations: $f(a) = \bot$ means $a \notin \mathsf{dom}(f)$, ϵ is the empty partial function with $\mathsf{dom}(\epsilon) = \varnothing$, $f[a \mapsto b]$ is the partial function that is equivalent to f except that $f(a) = b$, and $f{::}g$ is the partial function that is equivalent to g on $\mathsf{dom}(g)$ and equivalent to f on $A \setminus \mathsf{dom}(g)$. In our transition rules, we use patterns of the form $f(a_1 \mapsto b_1)\ldots(a_n \mapsto b_n)$ (where $n \geq 1$)) to match partial functions f that satisfy $f(a_i) = b_i$ for all $1 \leq i \leq n$.

3.1 Syntax and Semantics

We show the syntax and semantics of our calculus concisely in Fig. 5. The syntax (top left) represents a standard functional calculus, augmented with references. References can be created ($\mathsf{ref}\ e$), read ($!e$) and assigned ($e := e$). The result of a fork expression $\mathsf{rfork}\ e$ is a revision identifier from the set *Rid*, and can be used in a $\mathsf{rjoin}\ e$ expression (note that e is an expression, not a constant, thus the revision being joined can vary dynamically).

To define evaluation order within an expression, we syntactically define execution contexts (Fig. 5 right column, in the middle). An execution context $\mathcal{E}$ is an expression "with a hole $\square$", and as usual we let $\mathcal{E}[e]$ be the expression obtained from $\mathcal{E}$ by replacing the hole $\square$ with e.

The operational semantics (Fig. 5, bottom) describes transitions of the form $s \rightarrow_r s'$ which represent a step by

Syntactic Symbols

$$
\begin{array}{lclcl}
v & \in & \mathit{Val} & ::= & c \mid x \mid l \mid r \mid \lambda x.e \\
c & \in & \mathit{Const} & ::= & \mathsf{unit} \mid \mathsf{false} \mid \mathsf{true} \\
l & \in & \mathit{Loc} & & \\
r & \in & \mathit{Rid} & & \\
x & \in & \mathit{Var} & & \\
e & \in & \mathit{Expr} & ::= & v \\
 & & & \mid & e\ e \mid (e \mathbin{?} e : e) \\
 & & & \mid & \mathsf{ref}\ e \mid {!e} \mid e := e \\
 & & & \mid & \mathsf{rfork}\ e \mid \mathsf{rjoin}\ e
\end{array}
$$

State

$$
\begin{array}{lclcl}
s & \in & \mathit{GlobalState} & = & \mathit{Rid} \rightharpoonup \mathit{LocalState} \\
 & & \mathit{LocalState} & = & \mathit{Snapshot} \times \mathit{LocalStore} \times \mathit{Expr} \\
\sigma & \in & \mathit{Snapshot} & = & \mathit{Loc} \rightharpoonup \mathit{Val} \\
\tau & \in & \mathit{LocalStore} & = & \mathit{Loc} \rightharpoonup \mathit{Val}
\end{array}
$$

Execution Contexts

$$
\begin{array}{lcl}
\mathcal{E} & = & \square \\
 & \mid & \mathcal{E}\ e \mid v\ \mathcal{E} \mid (\mathcal{E} \mathbin{?} e : e) \\
 & \mid & \mathsf{ref}\ \mathcal{E} \mid {!\mathcal{E}} \mid \mathcal{E} := e \mid l := \mathcal{E} \\
 & \mid & \mathsf{rjoin}\ \mathcal{E}
\end{array}
$$

Operational Semantics

$$
\begin{array}{llcll}
(\mathit{apply}) & s(r \mapsto \langle \sigma, \tau, \mathcal{E}[(\lambda x.e)\ v] \rangle) & \rightarrow_r & s[r \mapsto \langle \sigma, \tau, \mathcal{E}[[v/x]e] \rangle] & \\
(\mathit{if\text{-}true}) & s(r \mapsto \langle \sigma, \tau, \mathcal{E}[(\mathsf{true} \mathbin{?} e_1 : e_2)] \rangle) & \rightarrow_r & s[r \mapsto \langle \sigma, \tau, \mathcal{E}[e_1] \rangle] & \\
(\mathit{if\text{-}false}) & s(r \mapsto \langle \sigma, \tau, \mathcal{E}[(\mathsf{false} \mathbin{?} e_1 : e_2)] \rangle) & \rightarrow_r & s[r \mapsto \langle \sigma, \tau, \mathcal{E}[e_2] \rangle] & \\
 & & & & \\
(\mathit{new}) & s(r \mapsto \langle \sigma, \tau, \mathcal{E}[\mathsf{ref}\ v] \rangle) & \rightarrow_r & s[r \mapsto \langle \sigma, \tau[l \mapsto v], \mathcal{E}[l] \rangle] & \text{if } l \notin s \\
(\mathit{get}) & s(r \mapsto \langle \sigma, \tau, \mathcal{E}[!l] \rangle) & \rightarrow_r & s[r \mapsto \langle \sigma, \tau, \mathcal{E}[(\sigma::\tau)(l)] \rangle] & \text{if } l \in \mathsf{dom}(\sigma::\tau) \\
(\mathit{set}) & s(r \mapsto \langle \sigma, \tau, \mathcal{E}[l := v] \rangle) & \rightarrow_r & s[r \mapsto \langle \sigma, \tau[l \mapsto v], \mathcal{E}[\mathsf{unit}] \rangle] & \text{if } l \in \mathsf{dom}(\sigma::\tau) \\
 & & & & \\
(\mathit{fork}) & s(r \mapsto \langle \sigma, \tau, \mathcal{E}[\mathsf{rfork}\ e] \rangle) & \rightarrow_r & s[r \mapsto \langle \sigma, \tau, \mathcal{E}[r'] \rangle][r' \mapsto \langle \sigma::\tau, \epsilon, e \rangle] & \text{if } r' \notin s \\
(\mathit{join}) & s(r \mapsto \langle \sigma, \tau, \mathcal{E}[\mathsf{rjoin}\ r'] \rangle)(r' \mapsto \langle \sigma', \tau', v \rangle) & \rightarrow_r & s[r \mapsto \langle \sigma, \tau::\tau', \mathcal{E}[\mathsf{unit}] \rangle][r' \mapsto \bot] & \\
(\mathit{join}_\epsilon) & s(r \mapsto \langle \sigma, \tau, \mathcal{E}[\mathsf{rjoin}\ r'] \rangle)(r' \mapsto \bot) & \rightarrow_r & \epsilon &
\end{array}
$$

Figure 5. Syntax and Semantics of the revision calculus.

revision r from global state s to global state s'. Consider first the definition of global states (Fig. 5, top right). A global state is a partial function from revision identifiers to local states: there is no shared global state. The local state has three parts (σ, τ, e): the snapshot σ is a partial function that represents the initial state that this revision started in, the local store τ is a partial function that represents all the locations this revision has written to, and e is the current expression.

The rules for the operational semantics (Fig. 5, bottom) all follow the same general structure: a transition $s \rightarrow_r s'$ matches the local state for r on the left, and describes how the next step of revision r changes the state.

The first three rules (*apply*), (*if-true*), and (*if-false*)) reflect standard semantics of application and conditional. They affect only the local expression. The next three rules (*new*), (*get*), and (*set*) reflect operations on the store. Thus, they affect both the local store and the local expression. The (*new*) rule chooses a fresh location (we simply write $l \notin s$ to express that l does not appear in any snapshot or local store of s). The last two rules reflect synchronization operations. The rule (*fork*) starts a new revision, whose local state consists of (1) a snapshot that is initialized to the current state $\sigma::\tau$, (2) a local store that is the empty partial function, and (3) an expression that is the expression supplied with the fork. Note that (*fork*) chooses a fresh revision identifier (we simply write $r \notin s$ to express that r is not mapped by s, and does not appear in any snapshot or local store of s). The rule (*join*) updates the local store of the revision that performs the join by merging the snapshot, master, and revision states (in accordance with the declared isolation types), and removes the joined revision. We call r the joining revision (or joiner), and r' the joined revision (or joinee). A join can only proceed if the joinee has executed all the way to a value (which is ignored). The final rule (join_ϵ) is added to prevent joining a revision handle more than once. If a revision handle is joined a second time, the joinee is no longer in the domain of s, and the entire state transitions to a special error state represented by the empty partial function ϵ (this state can not be reached in any other way, and has no outgoing transitions).

3.2 Executions

As usual, we let $\rightarrow$ be the union of all $\rightarrow_r$ where $r \in \mathit{Rid}$. Furthermore, we use the following notations for repeated steps: we say $s \rightarrow^n s'$ if s' can be reached from s in exactly n $\rightarrow$-steps, we say $s \rightarrow^* s'$ (transitive reflexive closure) if it can be reached in zero or more steps, $s \rightarrow^+ s'$ (transitive closure) if it can be reached in one or more steps, and $s \rightarrow^? s'$ (reflexive closure) if it can be reached in zero or one steps.

We define global executions of expressions as follows. First, we call an expression e a *program expression* if it does not contain any revision identifiers (expressions may contain revision identifiers during execution, but not initially). We say a sequence of transitions $s_0 \rightarrow s_1 \rightarrow \cdots \rightarrow s_n$ is an *execution* of a program expression e if $s_0 = \{(r, (\epsilon, \epsilon, e)\}$ for some $r \in \mathit{Rid}$. We call such an execution *maximal* if there exists no s' such that $s_n \rightarrow s'$. Finally, given a program expression e we write $e \downarrow s$ if there exists a maximal execution for e with final state s.

3.3 Determinacy

A surprising property of our calculus is that executions are determinate and not dependent on a specific ‘schedule’. Before we can state this precisely, we need a notion of equivalence of states modulo renaming of revisions and locations.

For a permutation α of *Rid* and a global state s let $\alpha(s)$ be the global state obtained by replacing all revision identifiers r that occur in s with $\alpha(r)$. Similarly, define $\beta(s)$ for a permutation β of *Loc*. We say two states s, s' are equivalent upto $\alpha\beta$-renaming, written as $s \approx s'$, if there exist permutations α of *Rid* and β of *Loc* such that $s = \alpha(\beta(s'))$.

We now state the main result of this section: executions are determinate modulo renaming of locations and revisions.

THEOREM 1 (Determinacy). *Let e be a program expression, and let $e \downarrow s$ and $e \downarrow s'$. Then $s \approx s'$.*

Before proving this theorem, we make a few observations, and establish a few lemmas and an important confluence theorem.

Note that some executions may terminate in the special error state ϵ if they attempt to join the same revision more than once. Our use of a special error state is important to guarantee determinacy. Suppose two revisions try to join a third revision simultaneously (i.e. there is a race between two joins). Without the rule (*join*$_\epsilon$) the different schedules may lead to different final states. However, with (*join*$_\epsilon$), all executions are forced to eventually end up at ϵ, maintaining determinacy.

To prepare for the proof, we now state and prove a local determinism lemma and a confluence theorem.

LEMMA 2 (Local Determinism). *If $s_1 \approx s'_1$ and $s_1 \rightarrow_r s_2$ and $s'_1 \rightarrow_r s'_2$, then $s_2 \approx s'_2$.*

PROOF. First we observe that by construction, each evaluation context $\mathcal{E}$ contains at most one hole and that there is no choice in which redex to evaluate next. We can now do a case analysis on $\mathcal{E}[e]$ where e is a redex. For a fixed revision r, such expression context is matched uniquely by at most one operational rule. Moreover, each rule is deterministic modulo $\alpha\beta$-equivalence. This is trivial for all operations except (*new*) and (*fork*) that create new locations and revisions respectively. Given a state $s(r \mapsto \langle \sigma, \tau, \mathcal{E}[\mathsf{ref}\ v] \rangle)$, rule (*new*) can create different names for the new location, i.e. $s = s(r \mapsto \langle \sigma, [\tau \mapsto l]v, \mathcal{E}[l] \rangle)$ or $s' = s(r \mapsto \langle \sigma, [\tau \mapsto l']v, \mathcal{E}[l'] \rangle)$. If $l = l'$ this is equivalent directly. If $l \neq l'$ we can apply α-renaming with $\alpha = [l/l']$ where $s = \alpha(s')$ which holds since $l' \notin s'$ and $l \notin s$ due to the side condition on (*new*) (and by definition $s \approx s'$). We prove equivalence similarly for (*fork*). □

LEMMA 3 (Strong Local Confluence). *Let s_1 and s'_1 be reachable states that satisfy $s_1 \approx s'_1$. Then, if $s_1 \rightarrow_r s_2$ and $s'_1 \rightarrow_{r'} s'_2$, then there exist equivalent states $s_3 \approx s'_3$ such that both $s_2 \rightarrow^?_{r'} s_3$ and $s'_2 \rightarrow^?_r s'_3$.*

PROOF. First we observe that when $r = r'$, the lemma follows directly from the local-determinism lemma. We continue the proof for the case $r \neq r'$, and do a case distinction on the kind of the two operational steps appearing in the assumption of the theorem. We use the term *local step* to denote a step that is not (*fork*), (*join*), and (*join*$_\epsilon$).

- (*local*) / (*local*). The rules affect independent parts of the state s and thus commute. As before, we may need to use α-renaming for the (*new*) case.
- (*local*) / (*fork*),(*join*). Same argument; note that the forked/joined revision can not be the same as the local one because of the side condition $r' \notin s$ (for fork) or because the joinee can not take a step (for join).
- (*join*$_\epsilon$) / any. The claim follows because if we could apply (*join*$_\epsilon$) in some state but perform a different rule, then (*join*$_\epsilon$) still applies.
- (*fork*) / (*fork*). In this case the side condition $r' \notin s$ ensures that both forks will fork a unique revision. As shown in the proof of the previous lemma, we can safely apply β-renaming to show both end states are equivalent.
- (*fork*) / (*join*). Observe that the (*join*) cannot join on the revision that forks (since its expression is not a value). Also, the side condition $r' \notin s$ ensures that a unique revision is forked that is different from r and r' in the (*join*) rules.
- (*join*) / (*join*). Consider the matched state for both rules: $s(r \mapsto \langle \sigma, \tau, \mathcal{E}[\mathsf{rjoin}\ r_1] \rangle)(r_1 \mapsto \langle \sigma_1, \tau_1, v_1 \rangle)$ and $s(r' \mapsto \langle \sigma', \tau', \mathcal{E}'[\mathsf{rjoin}\ r_2] \rangle)(r_2 \mapsto \langle \sigma_2, \tau_2, v_2 \rangle)$. We have two possiblities. First, if $r_1 \neq r_2$, both joins commute directly. Otherwise, $r_1 = r_2$. In this case the joinee is shared . Thus, taking step $\rightarrow_r$ leads to a state where $s(r_1 \mapsto \bot)$ and step $\rightarrow_{r'}$ must use (*join*$_\epsilon$) ending in state ϵ, which is also the outcome for the opposite order.

□

THEOREM 4 (Confluence). *For any reachable states $s_1 \approx s'_1$, it holds that if $s_1 \rightarrow^* s_2$ and $s'_1 \rightarrow^* s'_2$, then there exist equivalent states $s_3 \approx s'_3$ such that both $s_2 \rightarrow^* s_3$ and $s'_2 \rightarrow^* s'_3$.*

Proving confluence from strong local confluence is well-known and often illustrated using tiling of diagrams. It is useful for several applications (e.g. the lambda calculus or general term rewriting) but can also be understood more abstractly as a property of binary relations [18]. We include a quick proof sketch for reference.

PROOF. First, lift the step relation $\rightarrow$ to equivalence classes of states modulo $\approx$. Let x, y, z, u range over equivalence classes, and consider the following three properties:

1. $\forall xyz : x \rightarrow y \wedge x \rightarrow z \Rightarrow \exists u : y \rightarrow^? u \wedge z \rightarrow^? u$
2. $\forall n : \forall xyz : x \rightarrow^? y \wedge x \rightarrow^n z \Rightarrow \exists u : y \rightarrow^* u \wedge z \rightarrow^? u$
3. $\forall n : \forall xyz : x \rightarrow^n y \wedge x \rightarrow^* z \Rightarrow \exists u : y \rightarrow^* u \wedge z \rightarrow^* u$

We can then show that (1) the first claim follows from strong local confluence, (2) the second claim follows from the first by induction over n, (3) the third claim follows from the second by induction over n, and (4) the theorem follows from the third claim. □

We now conlude with the proof of theorem 1. Given a program expression e and two maximal executions $s_0 \rightarrow^* s$ and $s'_0 \rightarrow^* s'$ for e, we know $s_0 \approx s'_0$ (by the way we defined initial states for e), so by the confluence theorem there exist $s_1 \approx s'_1$ such that $s \rightarrow^* s_1$ and $s' \rightarrow^* s'_1$. But since s and s' are maximal it must be the case that $s = s_1$ and $s' = s'_1$ and thus $s \approx s'$ as claimed.

4. State Merging

The basic calculus introduced in the previous section provides little flexibility as to how write-write conflicts should be resolved. We now show how to modify the calculus so that it can support custom merge functions (Section 4.1), how it can be understood as an extension of snapshot isolation (Section 4.3), and how we can provide stronger consistency guarantees for abstract data types using sequential merge functions (Section 4.4).

4.1 Merge Functions

Figure 6 extends the basic calculus with flexible merge functions. There is just one change to the basic calculus where we replace the (*join*) rule with the (*join-merge*) rule. Instead of composing the new state as $\tau{::}\tau'$ we call a custom $\mathsf{merge}(\tau, \tau', \sigma')$ function that merges the states. If there is no (write-write) conflict at a particular location, this function

$$(join\text{-}merge) \quad s(r \mapsto \langle \sigma, \tau, \mathcal{E}[\mathsf{rjoin}\ r'] \rangle)(r' \mapsto \langle \sigma', \tau', v \rangle) \quad \rightarrow_r \quad s[r \mapsto \langle \sigma, \mathsf{merge}(\tau, \tau', \sigma'), \mathcal{E}[\mathsf{unit}] \rangle][r' \mapsto \bot]$$

$$\text{where} \qquad \mathsf{merge}(\tau, \tau', \sigma')(l) = \begin{cases} \tau(l) & \text{if } \tau'(l) = \bot \\ \tau'(l) & \text{if } \tau(l) = \bot \\ \mathsf{merge}_l(\tau(l), \tau'(l), \sigma'(l)) & \text{otherwise} \end{cases}$$

Figure 6. Extending the revision calculus with merge functions.

behaves just like our earlier composition. In case of conflict, the value at a location l after a join is determined by a location specific function $\mathsf{merge}_l : Val \times Val \times Val \rightarrow Val$ which is defined separately for each location l.

The merge_l function subsumes the semantics of the previous calculus where a joinee takes precedence since we can define the default merge function as:

$$\mathsf{merge}_l(v, v', v_0) = v' \quad \text{(joinee wins)}$$

Similarly, we can implement the dual strategy where updates to a specific location are ignored if there is a write-write conflict:

$$\mathsf{merge}_l(v, v', v_0) = v \quad \text{(joiner wins)}$$

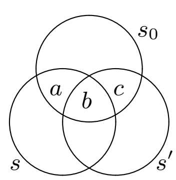
Note that sometimes, we may wish to define merge functions involving more than a single variable. In our calculus we can do so by using composite types to group several variables into a single location and merge them collectively.

4.2 Commutative Merges

We call a merge function *commutative* if $\mathsf{merge}_l(v, v', v_0) = \mathsf{merge}_l(v', v, v_0)$. Clearly, the default merge function is not commutative, but many others are. For example, a reasonable merge function for sets could be:

$$\mathsf{merge}_l(s, s', s_0) = s \cup s'$$

which is commutative. This is not the only reasonable merge function though. Consider the following venn diagram that shows how the sets s, s', and s_0 may interact:



When taking the union of s and s', we always include the regions a, b, and c. One can argue however that to end up with s' from s_0, the elements in a were explicitly removed (and similarly for s with region c). Another reasonable merge function may respect such removals and remove region a and c from the final result. We can specify this as:

$$\mathsf{merge}_l(s, s', s_0) = (s - s_0) \cup (s' - s_0) \cup (s \cap s')$$

which is also commutative. Note that when all operations on the set are additive, both of these merge functions produce the same result since $s_0 \subseteq (s \cap s')$ in that case.

Ultimately, this discussion simply illustrates that the choice of a merge function should be informed by what operations are performed (additions only, removals only, both, etc.). We discuss this idea more formally in Section 4.4, where we show that by restricting the operations on an abstract data type, we can find merge functions can provide particularly strong guarantees.

4.3 Snapshot isolation

We now explain how to view our system as a generalization of *snapshot isolation* [4], a multiversion concurrency control algorithm that is widely used in the database community, and has for example been implemented by Oracle and Microsoft SQL Server (with minor variations). We use the definition given by Fekete *et al.* [11].

We claim that our revision calculus is a generalization of snapshot isolation, augmented by (1) the ability to gracefully resolve write-write conflict when a suitable merge function exists for a particular location, and (2) support nontrivial nesting (Fig. 3) while maintaining a simple and precise semantics. To see why this is the case, we perform the reverse process: we (1) introduce the ability to fail on write-write conflicts, and (2) remove nesting from revisions.

Removing nesting is straightforward (for example, we can disallow forks by all revisions but the main revision). As for failing on conflicts, we proceed as follows. To mirror how transactions fail (and discard state), we introduce the notion of a failing join as follows.

- We change the merge calculus slightly, by redefining the local merge functions to that they can return a special value indicating that there is an unresolvable conflict:

$$\mathsf{merge}_l : Val \times Val \times Val \rightarrow (Val \cup \{\mathsf{undef}\})$$

- We extend the merge calculus by replacing (*join-merge*) with two new rules. The (*join-ok*) rule is equivalent to the previous (*join-merge*) rule but can only be applied now if all of the location specific merge functions are defined. The rule (*join-fail*) applies if at least one of the merges failed and simply ignores all updates in the joinee. Both rules now return a boolean to the joiner, where true indicates that the join was successful, and false indicates that it was not.

Consider now the definition of snapshot isolation: A transaction A executing under snapshot isolation operates on a snapshot of the database taken at the start of the transaction. When the transaction concludes, it will successfully commit only if the values updated by the transaction A were not updated by any other transaction B that committed after transaction A started.

We can succinctly describe this behaviour in our calculus by letting every merge_l function fail:

$$\mathsf{merge}_l(v, v', v_0) = \mathsf{undef} \qquad \text{(snapshot isolation)}$$

When discussing snapshot isolation there is sometimes confusion whether a transaction should abort if there was a concurrent *silent write* where original value has been left unchanged. In our formal calculus there is of course no confusion (and concurrent silent writes will indeed cause a transaction to fail). However, we can also concisely describe this behaviour as well:

$$\mathsf{merge}_l(v, v', v_0) = ((v = v_0)\ ?\ v' : \mathsf{undef}) \quad \text{(ignore silent wr)}$$

$$
\begin{array}{llll}
(\textit{join-ok}) & s(r \mapsto \langle \sigma, \tau, \mathcal{E}[\mathsf{rjoin}\ r'] \rangle)(r' \mapsto \langle \sigma', \tau', v \rangle) \rightarrow_r s[r \mapsto \langle \sigma, \mathsf{merge}(\tau, \tau', \sigma'), \mathcal{E}[\mathsf{true}] \rangle][r' \mapsto \bot] & \text{if } \neg\mathsf{fail}(\tau, \tau', \sigma') \\
(\textit{join-fail}) & s(r \mapsto \langle \sigma, \tau, \mathcal{E}[\mathsf{rjoin}\ r'] \rangle)(r' \mapsto \langle \sigma', \tau', v \rangle) \rightarrow_r s[r \mapsto \langle \sigma, \tau, \mathcal{E}[\mathsf{false}] \rangle][r' \mapsto \bot] & \text{if } \mathsf{fail}(\tau, \tau', \sigma') \\
\text{where} & \mathsf{fail}(\tau, \tau', \sigma') = \mathsf{undef} \in \mathsf{rg}(\mathsf{merge}(\tau, \tau', \sigma')) &
\end{array}
$$

Figure 7. Extending the merge calculus with failing joins.

4.4 Abstract Data Types and Sequential Merges

As Theorem 1 shows, our calculus is always determinate, but we have seen in the introduction that it is not always serializable (Fig. 2). However, we can sometimes guarantee equivalence to a sequential execution by raising the abstraction level of operations on data, and constructing merge functions that tailored to the operations that performed.

For example, consider a program location x that is initially zero and for which we define the merge function $\mathsf{merge}_x(v, v', v_0) = v + v' - v_0$. Furthermore, assume that a program performs only one type of operation on x, namely $\mathit{add}(i)$, which adds an integer i to it. Then the final value of x is always consistent with a serial execution of all the *add* operations that occurred in the program. We now explain this idea more formally.

4.4.1 Abstract Data Types

We define an *abstract data type* to be a tuple $(V, o, \mathit{Op}, \mathit{op})$ where V is a set of values,$o \in V$ is an initial value, Op is a set of operations, and $\mathit{op} : \mathit{Op} \times V \rightharpoonup V$ is a partial function. In our formalization, the set Op includes argument and return values of operations, and op is partial because not all operations apply in all states.

EXAMPLE 5. *We can define an integer register (i.e. a memory location holding an integer that can be read and written) as* $\mathit{IntReg} = (\mathbb{Z}, 0, \mathit{Op}, \mathit{op})$ *where*

$$
\begin{array}{rcl}
\mathit{Op} & = & \{\mathit{get}(v) \mid v \in \mathbb{Z}\} \cup \{\mathit{set}(v) \mid v \in \mathbb{Z}\} \\
\mathit{op}(v, o) & = & \begin{cases} w & \text{if } o = \mathit{set}(w) \\ v & \text{if } o = \mathit{get}(v) \\ \bot & \text{if } o = \mathit{get}(v') \text{ and } v \neq v' \end{cases}
\end{array}
$$

4.4.2 Sequential Merge Functions

Sometimes we can find merge functions that can simulate a deterministic, linear interleaving of the operations. We call such merge functions *sequential*. This concept is quite useful in practice since the programmer can design the application specifically to enable sequential merge functions, by restricting what type of operations may happen in concurrent revisions. For example, if an application performs aggregation of results, sequential merges usually exist.

To study the effect of entire sequences of operations, we introduce the following concise notations. We consider operation sequences as words in Op^*, and write $u(v)$ (where $u \in \mathit{Op}^*$ and $v \in \mathit{Val}$) for the combined effect of all the operations in the sequence u (left to right) applied to the value v, which may be undefined. For example, this means that for operation sequences $u, w \in \mathit{Op}^*$ and a value v, we have $uw(v) = w(u(v))$ if $u(v) \neq \bot$ and $w(u(v)) \neq \bot$. We now define sequential merge functions as follows.

DEFINITION 6. *Let* $\mathcal{A} = (V, o, \mathit{Op}, \mathit{op})$ *be an abstract data type. We say a merge function* $\mathsf{m} : V \times V \times V \to V$ *is* sequential *for* $\mathcal{A}$ *if for all operation sequences* $u, w_1, w_2 \in \mathit{Op}^*$ *such that* $u(o) \neq \bot$, $uw_1(o) \neq \bot$ *and* $uw_2(o) \neq \bot$, *both of the following are true:*

1. $uw_1w_2(o) \neq \bot$
2. $\mathsf{m}(uw_1(o), uw_2(o), u(o)) = uw_1w_2(o)$

The advantage of a sequential merge function is that it guarantees the appearance that all operations were executed sequentially, with the operations of the joined revision happening at the time of the join.

Note that condition 1 of Def. 6 does not depend on the actual merge function, but is a property of the abstract data type. This property may not be satisfiable, thus sequential merge functions do not exist for all abstract data types. For example, the abstract data type *IntReg* defined in Example 5 does not permit sequential merging because it can be the case that $w_1 = \mathit{set}(1)$, $w_2 = \mathit{get}(0)$ in which case always $uw_1w_2(0) = \bot$.

4.4.3 Abelian Data Types

Particularly simple to merge are certain abstract data types with commutative operations that we call *abelian*. More formally, call an abstract data type $(V, o, \mathit{Op}, \mathit{op})$ *abelian* if there exists a binary operation $+$ on V, and a function $\delta : \mathit{Op} \to V$ such that (1) $(V, +)$ is an abelian group with neutral element o, and (2) for all $a \in \mathit{Op}$ we have $a(v) = v + \delta(a)$.

We conclude this section with a lemma that shows how to construct sequential merge functions for abelian data types.

LEMMA 7. *For an abelian data type* $(V, o, \mathit{Op}, \mathit{op})$ *with operation* $+$*, the merge function :* $\mathsf{m}(v_1, v_2, v) = v_1 + v_2 - v$ *is sequential.*

PROOF. Let $w_1 = a_1 \ldots a_n$ and $w_2 = b_1 \ldots b_m$ and $v = u(o)$. Then claim 2 is satisfied: $\mathsf{m}(uw_1(o), uw_2(o), u(o)) = \mathsf{m}(w_1(u(o)), w_2(u(o)), u(o)) = \mathsf{m}(v + a_1 + \cdots + a_n, v + b_1 + \cdots + b_m, v) = v + a_1 + \cdots + a_n + v + b_1 + \cdots + b_m - v = v + a_1 + \cdots + a_n + b_1 + \cdots + b_m = w_1w_2(u(o)) = uw_1w_2(o)$. This implies also that claim 1 is satisfied. □

5. Revision Diagrams

In this section we describe and formally define *revision diagrams*, a special kind of graph that visually represent the dataflow of computations of our calculus. We then prove the main result of this section: that revision diagrams are semi-lattices. This result illuminates how revision graphs differ from task-parallel models that allow arbitrary directed acyclic graphs of tasks. Moreover, the existence of a greatest common ancestor for any two states helps to reason about state merging.

Intuitively, revision diagrams represent executions, with vertices being states and edges being transitions. Technically, revision diagrams are labeled graphs:

DEFINITION 8. *A* fsj-graph G *is a tuple* $G = (V, E)$ *where* V *is a set of vertices and* $E \subset V \times \{\mathsf{f}, \mathsf{s}, \mathsf{j}\} \times V$ *is a set of labeled edges.*

Graph Notations. We use the usual terminology for graphs, but emphasize a relational view of edges. For a fixed

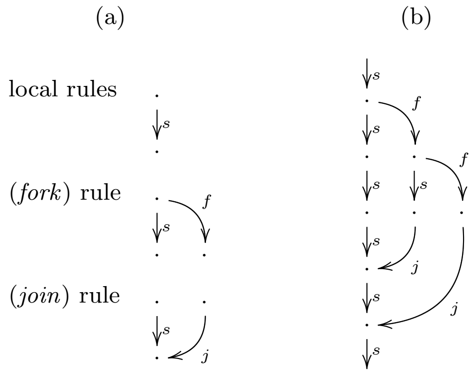


Figure 8. (a) Illustration of how revision diagrams are constructed incrementally by transition rules adding vertices. (b) An example of a revision diagram.

graph $G = (V, E)$, we define a binary relation $\xrightarrow{f}$ on vertices such that $(u \xrightarrow{f} v) \overset{\text{def}}{\Leftrightarrow} ((u, f, v) \in E)$, and similarly for $\xrightarrow{j}$ and $\xrightarrow{s}$. We also define the relation $\rightarrow \overset{\text{def}}{=} (\xrightarrow{f} \cup \xrightarrow{j} \cup \xrightarrow{s})$. Furthermore, we sometimes compose the above relations or take their Kleene closure, using the usual notation. For example, for two vertices u and v, the statement $u \xrightarrow{f}\rightarrow^* v$ means that there exists a path from u to v starting with a f-edge.

5.1 Operational Construction

To define the revision diagram for a given execution, we extend the original operational semantics so that a fsj-graph is constructed alongside the executing program. More formally, we extend the original transition relation $\rightarrow$ to an extended transition relation $\rightarrow_d$ (where d is just a label for easier distinction) on states $(s, (V, E), \rho, \gamma, o)$ where

- $s \in \mathit{GlobalState}$ is a global state as defined previously.
- (V, E) is a fsj-graph.
- $\rho : V \rightarrow \mathit{Rid}$ maps vertices to revisions.
- $\gamma : \mathit{Rid} \rightharpoonup V$ is a partial function that maps a revision to the last (current) vertex of that revision.
- $o : \mathit{Rid} \rightharpoonup V$ is a partial function that maps each revision to the vertex from which it forked.

The graph is constructed incrementally by adding new vertices, and edges from the existing graph to the new vertices, as illustrated in Fig. 8 (a). The precise transition rules for $\rightarrow_d$ are defined in Fig. 9. Intutively, the constructed graphs represents excecutions in the following sense (for an example, see Fig. 8 (b)):

- Each vertex $v \in V$ belongs to a particular revision $\rho(v)$ and represents the local state of that revision at a certain point of time.
- The set of all vertices belonging to the same revisions is totally ordered by $\xrightarrow{s}$, the successor relation, which describes how the local state of that revision evolves over time.
- Each edge in $\xrightarrow{f}$ represents the forking of a new revision. Its destination vertex is the first vertex of the new revision.
- Each edge in $\xrightarrow{j}$ represents the joining of a revision. Its source vertex is that last vertex of the revision being joined.

We define the initial states to be $(s_0, G_0, \rho_0, \gamma_0, o_0)$ where $s_0 = (r, (\epsilon, \epsilon, e))$ is an initial global state (that is, r is a revision identifier and e is an expression not containing any revision identifiers), $G_0 = (\{v\}, \emptyset)$ is a singleton graph, $\rho_0 = \{v \mapsto r\}$, $\gamma_0 = \{r \mapsto v\}$, and $o = \emptyset$ (because the initial revision was not forked). We say a state (s, G, ρ, γ, o) is *reachable* if there exists an initial state from which it can be reached by zero or more $\rightarrow_d$-transitions.

Definition 9. *A* revision diagram *is an fsj-graph G that is part of some reachable state (s, G, ρ, γ, o).*

It is easy to see (comparing the definitions of $\rightarrow$ and $\rightarrow_d$) that for any execution $s_0 \rightarrow^* s_n$ such that $s_n \neq \epsilon$, we can find a corresponding extended execution, and vice versa. Note that this may entail the renaming of revision identifiers in the $\rightarrow$-execution, because the latter does allow the reuse of revision identifiers after the revision has been joined while $\rightarrow_d$ does not.

Our main result of this section is the following:

Theorem 10. *Let $G = (V, E)$ be a revision diagram. Then G is a semilattice, i.e. for any two vertices $x, y \in V$, there exists a greatest common ancestor.*

Before proving this theorem in Section 5.4, we prepare by establishing some basic insight about the structure of revision diagrams.

5.2 Structure and Layout

We now state and prove some graph properties of revision diagrams. We say a vertex v is an ancestor of a vertex w if $v \rightarrow^* w$. The following two lemmas illustrate the structure of revision diagrams. Both can be easily proved by induction (detailed proofs are in the appendix).

Lemma 11. *Let $G = (V, E)$ be a revision diagram. Then G is acyclic, and contains a vertex that is an ancestor of all vertices (called the root).*

Lemma 12. *Let $G = (V, E)$ be a revision diagram, and let $v \in V$ be a vertex. Then the list of outgoing edges of v is either $\emptyset$, $\{\xrightarrow{s}\}$, $\{\xrightarrow{j}\}$, or $\{\xrightarrow{s}, \xrightarrow{f}\}$, and the list of incoming edges of v is either $\emptyset$, $\{\xrightarrow{s}\}$, $\{\xrightarrow{f}\}$, or $\{\xrightarrow{s}, \xrightarrow{j}\}$.*

By construction, the vertices belonging to a particular revision form a $\xrightarrow{s}$-sequence (see Fig. 8(b) for an example). When we draw revision diagrams, we follow the convention of drawing $\xrightarrow{s}$ chains vertically. Moreover, we often draw $\xrightarrow{f}$ edges curved, down and from left to right, and we draw $\xrightarrow{j}$ edges curved, down and from right to left.

The following theorem shows that it is indeed always possible to arrange revisions horizontally in that manner, because there exists a partial order on revisions such that if r forks r' or r joins r', then $r < r'$. This order turns out to be important to prove the semilattice theorem later on. We include the proof in the appendix.

Theorem 13 (Layout Order). *Let $(s, (V, E), \rho, \gamma, o)$ be a reachable state. Then there exists a partial order $<$ on Rid (called a* layout order*) such that for all vertices $v, w \in V$, the following two statements are true: (1) $v \xrightarrow{f} w$ implies $\rho(v) < \rho(w)$, and (2) $v \xrightarrow{j} w$ implies $\rho(w) < \rho(v)$.*

$$
\begin{array}{ll}
(\mathit{apply}) & (s(r \mapsto \langle \sigma, \tau, \mathcal{E}[(\lambda x.e)\ v]\rangle),\ (V,E), \rho, \gamma, o\ ) \quad \rightarrow_d \quad (\text{if } v \notin V) \\
& (s[r \mapsto \langle \sigma, \tau, \mathcal{E}[[v/x]e]\rangle],\ (V \cup v, E \cup \gamma(r) \xrightarrow{s} v), \rho[v \mapsto r], \gamma[r \mapsto v], o\ ) \\
(\mathit{if\text{-}true}) & (s(r \mapsto \langle \sigma, \tau, \mathcal{E}[(\mathsf{true}\,?\,e_1 : e_2)]\rangle),\ (V,E), \rho, \gamma, o\ ) \quad \rightarrow_d \quad (\text{if } v \notin V) \\
& (s[r \mapsto \langle \sigma, \tau, \mathcal{E}[e_1]\rangle],\ (V \cup v, E \cup \gamma(r) \xrightarrow{s} v), \rho[v \mapsto r], \gamma[r \mapsto v], o\ ) \\
(\mathit{if\text{-}false}) & (s(r \mapsto \langle \sigma, \tau, \mathcal{E}[(\mathsf{false}\,?\,e_1 : e_2)]\rangle),\ (V,E), \rho, \gamma, o\ ) \quad \rightarrow_d \quad (\text{if } v \notin V) \\
& (s[r \mapsto \langle \sigma, \tau, \mathcal{E}[e_2]\rangle],\ (V \cup v, E \cup \gamma(r) \xrightarrow{s} v), \rho[v \mapsto r], \gamma[r \mapsto v], o\ ) \\
(\mathit{new}) & (s(r \mapsto \langle \sigma, \tau, \mathcal{E}[\mathsf{ref}\ v]\rangle),\ (V,E), \rho, \gamma, o\ ) \quad \rightarrow_d \quad (\text{if } l \notin s \text{ and } v \notin V) \\
& (s[r \mapsto \langle \sigma, \tau[l \mapsto v], \mathcal{E}[l]\rangle],\ (V \cup v, E \cup \gamma(r) \xrightarrow{s} v), \rho[v \mapsto r], \gamma[r \mapsto v], o\ ) \\
(\mathit{get}) & (s(r \mapsto \langle \sigma, \tau, \mathcal{E}[!l]\rangle),\ (V,E), \rho, \gamma, o\ ) \quad \rightarrow_d \quad (\text{if } l \in \mathsf{dom}(\sigma{::}\tau) \text{ and } v \notin V) \\
& (s[r \mapsto \langle \sigma, \tau, \mathcal{E}[(\sigma{::}\tau)(l)]\rangle]\ (V \cup v, E \cup \gamma(r) \xrightarrow{s} v), \rho[v \mapsto r], \gamma[r \mapsto v], o\ ) \\
(\mathit{set}) & (s(r \mapsto \langle \sigma, \tau, \mathcal{E}[l := v]\rangle),\ (V,E), \rho, \gamma, o\ ) \quad \rightarrow_d \quad (\text{if } l \in \mathsf{dom}(\sigma{::}\tau) \text{ and } v \notin V) \\
& (s[r \mapsto \langle \sigma, \tau[l \mapsto v], \mathcal{E}[\mathsf{unit}]\rangle],\ (V \cup v, E \cup \gamma(r) \xrightarrow{s} v), \rho[v \mapsto r], \gamma[r \mapsto v], o) \\
(\mathit{fork}) & (s(r \mapsto \langle \sigma, \tau, \mathcal{E}[\mathsf{rfork}\ e]\rangle)\ (V,E), \rho, \gamma, o\ ) \quad \rightarrow_d \quad (\text{if } r' \notin s \text{ and } v, w \notin V \ \text{ and } r' \notin \mathsf{rg}(\rho)\ ) \\
& (s[r \mapsto \langle \sigma, \tau, \mathcal{E}[r']\rangle][r' \mapsto \langle \sigma{::}\tau, \epsilon, e\rangle], \\
& \qquad (V \cup \{v,w\}, E \cup \{\gamma(r) \xrightarrow{s} v, \gamma(r) \xrightarrow{f} w\}), \rho[v \mapsto r], \rho[w \mapsto r'], \gamma[r \mapsto v][r' \mapsto w], o[r' \mapsto \gamma(r)]\ ) \\
(\mathit{join}) & (s(r \mapsto \langle \sigma, \tau, \mathcal{E}[\mathsf{rjoin}\ r']\rangle)(r' \mapsto \langle \sigma', \tau', v\rangle)\ (V,E), \rho, \gamma, o\ ) \quad \rightarrow_d \quad (\text{if } v \notin V) \\
& (s[r \mapsto \langle \sigma, \tau{::}\tau', \mathcal{E}[\mathsf{unit}]\rangle][r' \mapsto \bot],\ (V \cup v, E \cup \{\gamma(r) \xrightarrow{s} v, \gamma(r') \xrightarrow{j} v\}), \rho[v \mapsto r], \gamma[r \mapsto v][r' \mapsto \bot], o[r' \mapsto \bot]\ )
\end{array}
$$

Figure 9. Operational rules for $\rightarrow_d$.

Note that revision diagrams are not necessarily planar (i.e. sometimes there must be some crossing edges in any layout).

5.3 Direct Paths

We now continue to gather ammunition needed for the proof of Thm. 10 by studying properties of paths in revision diagrams. First, we define the notion of path formally. Then we talk about certain types of paths.

Definition 14. *Given a revision diagram $G = (V,E)$, we define a* path *p from vertex v to vertex w to be a tuple $(v_1, \ldots, v_n)$ of length at least one ($n \geq 1$) such that*

$$v = v_1 \rightarrow v_2 \ldots \rightarrow v_n = w.$$

We write $x \overset{f}{\twoheadrightarrow} y$ if there exists a path from x to y that contains at least one f-edge and no j-edges. Similarly, we write $x \overset{j}{\twoheadrightarrow} y$ if there exists a path from x to y that contains at least one j-edge and no f-edges. We call a path *direct* if all of its f-edges (if any) appear after all of its j-edges (if any).

The following theorem shows that there always exists a direct path between connected vertices:

Theorem 15. *Let (V,E) be a revision diagram and $x, y \in V$. If $x \rightarrow^* y$, then either $x \xrightarrow{s}{}^* y$, or $x \overset{f}{\twoheadrightarrow} y$, or $x \overset{j}{\twoheadrightarrow} y$, or $x \overset{j}{\twoheadrightarrow}\overset{f}{\twoheadrightarrow} y$.*

Proof. For a path p, let $w(p) \subset \{f, s, j\}^*$ be the word representing the sequence of edge labels, and let $<$ be the lexicographic order on words induced by the total order $s < j < f$ on labels (that is, $w_1 < w_2$ if either w_1 is a prefix of w_2, or there exists an index k such that $w_1[i] = w_2[i]$ for $i < k$ and either $w_1[k] < w_2[k]$). Now assume the claim is false and consider a path $p = (p_0, p_1, \ldots, p_n)$ from x to y such that $w(p)$ is minimal with respect to $<$. Since the claim is false there must exist $i < j$ such that

$$x = p_0 \rightarrow^* p_i \xrightarrow{f} p_{i+1} \xrightarrow{s}{}^* p_j \xrightarrow{j} p_{j+1} \rightarrow^* p_n = y.$$

By Lemma 16 (stated below) there must thus exist a path p' along the lines of

$$x = p_0 \rightarrow^* p_i \xrightarrow{s}\rightarrow^* \xrightarrow{s} p_{j+1} \rightarrow^* p_n = y.$$

Now p' can not be direct because we assumed the claim is false. But then it contradicts the minimality of p because $w(p') < w(p)$. ☐

The following lemma captures the intuition that there must exist a path from where a revision is forked to where it is joined that starts with an s-edge (this is because the handle must flow along some such path). The proof is a bit technical, so it resides in the appendix.

Lemma 16. *Let (V,E) be a revision diagram, and let $x, y \in V$. If $x \xrightarrow{f} (\xrightarrow{s})^* \xrightarrow{j} y$ then $x \xrightarrow{s} (\rightarrow)^* \xrightarrow{s} y$.*

5.4 Greatest Common Ancestors

We are now ready to prove the key lemma about uniqueness of maximal common ancestors.

Lemma 17. *Let (V,E) be a revision diagram, then the following property holds for all pairs of vertices $a, b \in V$: there do not exist two distinct vertices $v, w \in V$ that are both maximal common ancestors of a and b.*

Proof. We assume that the lemma does not hold and show that a contradiction results. First, let us choose a minimal

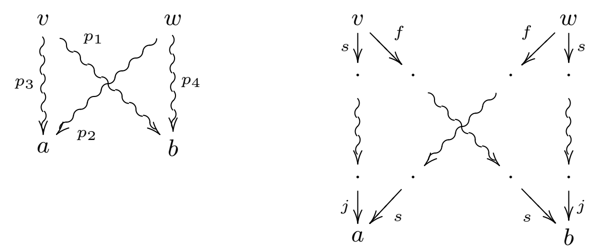


Figure 10. Illustration of the paths used in the proof of Lemma 17.

pair $(a, b) \in V$ violating the property (meaning that there exists no ancestor a' of a such that the property is violated for (a', b), nor an analogous ancestor b' of b). Because we assume the property is violated there exist distinct maximal common ancestors v, w of a, b. Note that $v, w \notin \{a, b\}$ because that would imply that there is a path from v to w or vice versa which would contradict maximality. Also, it can not be the case that $a = b$, otherwise maximality implies $a = b = v = w$. Thus a, b, v, w must be four distinct vertices connected by four paths p_1, p_2, p_3, p_4 as shown in Fig. 10(a). By Theorem 15, we can assume that all of those paths are direct. We now proceed by reasoning about these paths. Maximality of v implies that p_3, p_1 do not start with the same edge, thus they must start with an f-edge and a s-edge, respectively (Lemma 12). Without loss of generality, assume p_1 starts with an f-edge (otherwise switch a and b). Minimality of a, b implies that the two paths p_1, p_4 do not have the same last edge (if they share the last edge then an ancestor b' of b exists such that v and w are maximal common ancestors of a, b'), so those edges must be an s-edge and a j-edge (Lemma 12). Because p_1 starts with an f-edge, it can not contain any j-edges (remember that it is direct), so its last edge must be an s-edge, and p_4 must end with a j-edge, so it can not contain any f-edges. Thus $v \overset{f}{\twoheadrightarrow} b$ and $w \overset{j}{\twoheadrightarrow} b$. Symmetrically, we can reason about p_3 and p_2 and get $w \overset{f}{\twoheadrightarrow} a$ and $v \overset{j}{\twoheadrightarrow} a$ (Fig. 10(b)). But this is impossible because of the layout order theorem (Theorem 13), because it would imply a cycle $\rho(v) < \rho(b) < \rho(w) < \rho(a) < \rho(v)$ in the layout order. □

Of specific interest is the greatest common ancestor of the two vertices joined by a join transition, which is the vertex from which the joined transition was forked:

LEMMA 18. *Let* $(s, (V, E), \rho, \gamma, o) \rightarrow_d (s', (V', E'), \rho', \gamma', o')$ *be a join transition as in Fig. 9. Then* $o(r')$ *is the greatest common ancestor of* $\gamma(r)$ *and* $\gamma(r')$.

The proof is similar to the previous one and can be found in the appendix.

5.4.1 Proof of Thm. 10

This proof is now simple. For given vertices $x, y \in V$, the number of common ancestors is finite (because V is finite) and not zero (by Lemma 11). Thus there must exist some maximal common ancestors, and thus by Lemma 17 a greatest common ancestor.

6. Conclusion and Future Work

We have presented a novel programming model based on concurrent revisions. First, we presented a concise calculus that shows how revisions can maintain determinacy despite nondeterministic scheduling. Then we provided a discussion of how state merging can be tailored to the needs of the application. Finally, we formalized revision diagrams as the fundamental tool to visualize nonlinear histories of state, and showed graph properties that distinguish revision diagrams from general task graphs.

In future work, we may further investigate state merging and serialization guarantees. We are also interested in enhancing the calculus with reactive inputs and outputs and extending the determinacy guarantee to such applications. Finally, the precise characterization of revision diagrams remains an open problem. Last but not least, we are working on improving our library implementation and applying it to parallelize more applications.

Acknowledgments

We thank Tim Harris and Tom Ball for helpful comments and discussions.

References

[1] S. Aditya, Arvind, L. Augustsson, J.-W. Maessen, and R. Nikhil. Semantics of pH: A Parallel Dialect of Haskell. In Paul Hudak, editor, *Proc. Haskell Workshop, La Jolla, CA USA*, pages 35–49, June 1995.

[2] E. Allen, D. Chase, C. Flood, V. Luchangco, J.-W. Maessen, S. Ryu, and G. Steele Jr. Project fortress: A multicore language for multicore processors. In *Linux Magazine*, September 2007.

[3] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *Proceedings of SIGMOD*, pages 1–10, 1995.

[4] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ansi sql isolation levels. In *SIGMOD '95: International Conference on Management of Data*, pages 1–10, 1995.

[5] E. Berger, T. Yang, T. Liu, and G. Novark. Grace: Safe multithreaded programming for c/c++. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2009.

[6] G. Blelloch, S. Chatterjee, J. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, April 1994.

[7] R. Bocchino, V. Adve, D. Dig., S. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel java. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2009.

[8] S. Burckhardt, A. Baldassin, and D. Leijen. Concurrent programming with revisions and isolation types. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 2010.

[9] S. Burckhardt and D. Leijen. Semantics of concurrent revisions (full version). Technical Report MSR-TR-2010-94, Microsoft, 2010.

[10] P. Denning and J. Dennis. The resurgence of parallelism. *Commun. ACM*, 53(6), 2010.

[11] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2):492–528, 2005.

[12] C. Flanagan and M. Felleisen. The semantics of future and its use in program optimization. In *Rice University*, pages 209–220, 1995.

[13] M. Frigo, P. Halpern, C. E. Leiserson, and S. Lewin-Berlin. Reducers and other cilk++ hyperobjects. In *Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 79–90, 2009.

[14] M. Frigo, C. Leiserson, and K. Randall. The implementation of the Cilk-5 multithreaded language. In *Programming Language Design and Impl. (PLDI)*, pages 212–223, 1998.

[15] T. Harris, A. Cristal, O. Unsal, E. Ayguadé, F. Gagliardi, B. Smith, and M. Valero. Transactional memory: An overview. *IEEE Micro*, 27(3):8–29, 2007.

[16] M. Herlihy and E. Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *Principles and Practice of Parallel Programming (PPoPP)*, pages 207–216, 2008.

[17] M. Herlihy and J. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[18] G. Huet. Confluent reductions: Abstract properties and applications in term rewriting systems. *J. ACM*, 27(4), October 1980.

[19] E. Koskinen, M. Parkinson, and M. Herlihy. Coarse-grained transactions. In *Principles of Programming Languages (POPL)*, pages 19–30, 2010.

[20] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. Chew. Optimistic parallelism requires abstractions. In *Programming Language Design and Impl. (PLDI)*, 2007.

[21] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comp.*, C-28(9):690–691, 1979.

[22] J. Larus and R. Rajwar. *Transactional Memory*. Morgan & Claypool Publishers, 2007.

[23] J. Lee and J. Palsberg. Featherweight x10: a core calculus for async-finish parallelism. In *Principles and Practice of Parallel Programming (PPoPP)*, 2010.

[24] A. Martin, A. Birrell, T. Harris, and M. Isard. Semantics of transactional memory and automatic mutual exclusion. In *Principles of Programming Languages (POPL)*, pages 63–74, 2008.

[25] L. Moreau. The semantics of scheme with future. In *In In ACM SIGPLAN International Conference on Functional Programming (ICFP'96*, pages 146–156, 1996.

[26] P.A.Bernstein and N.Goodman. Multiversion concurrency control—theory and algorithms. *ACM Trans. Database Syst.*, 8(4):465–483, 1983.

[27] P.A.Bernstein, V.Hadzilacos, and N.Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[28] P. Pratikakis, J. Spacco, and M. Hicks. Transparent proxies for java futures. *SIGPLAN Not.*, 39(10):206–223, 2004.

[29] K. Randall. *Cilk: Efficient Multithreaded Computing*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1998.

[30] T. Riegel, C. Fetzer, and P. Felber. Snapshot isolation for software transactional memory. In *Workshop on Transactional Computing (TRANSACT)*, 2006.

[31] G. Steele. Parallel programming and parallel abstractions in fortress. In *Invited talk at the Eighth International Symposium on Functional and Logic Programming (FLOPS)*, April 2006.

[32] A. Welc, S. Jagannathan, and A. Hosking. Safe futures for java. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 439–453, 2005.

[33] A. Welc, B. Saha, and A.-R. Adl-Tabatabai. Irrevocable transactions and their applications. In *Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 285–296, 2008.

A. Proofs

A.1 Proof of Lemma 11

By induction. If G is part of an initial state, the claim is immediate because $G_0 = (\{v\}, \emptyset)$. Otherwise, let $(s', G', \rho', \gamma', o') \rightarrow_d (s, G, \rho, \gamma, o)$ be the last transition. By the induction hypothesis, G' is acyclic and contains a vertex that is an ancestor of all vertices. Because transitions change the graph only by (1) adding one or two new vertices and (2) adding edges from the old graph to the new vertices, G is also acyclic and all of its vertices are reachable from the root.

A.2 Proof of Lemma 12

We prove the following slightly stronger characterization by induction: (1) v has either two incoming edges ($\xrightarrow{s}$ and $\xrightarrow{j}$), one incoming edge ($\xrightarrow{s}$ or $\xrightarrow{f}$) or no incoming edges (if and only if v is the root), and (2) v has either two outgoing edges ($\xrightarrow{f}$ and $\xrightarrow{s}$), one outgoing edge ($\xrightarrow{s}$ or $\xrightarrow{j}$) or no outgoing edges (if and only if $v \in \mathsf{rg}(\gamma)$).

To prove statement (1), consider that incoming edges are added to a vertex only when the vertex is first added to the graph. The claim follows from examining the initial states (no incoming edge) and the added vertices (two incoming edges for the join transition, one incoming edge for all the others). To prove statement (2), consider that a new vertex has initially no outgoing edges and is in $\mathsf{rg}(\gamma)$ (which contains vertices with no outgoing edges only). A transition may add add either two (fork) or one (all other transitions) outgoing edge to some vertex in $\mathsf{rg}(\gamma)$, and removes it from $\mathsf{rg}(\gamma)$ in the process.

A.3 Proof of Theorem 13

We prove this by constructing an acyclic binary relation $\rightarrow_l$ on revisions such that $v \xrightarrow{f} w$ implies $\rho(v) \rightarrow_l \rho(w)$ and $v \xrightarrow{j} w$ implies $\rho(w) \rightarrow_l \rho(v)$; once we have such a relation we can easily obtain a suitable order by defining $< \stackrel{\text{def}}{=} \rightarrow_l^+$.

We construct $\rightarrow_l$ alongside the execution, starting empty, and adding edges for fork and join transitions as follows: (1) for a fork transition, add $r \rightarrow_l r'$, and also add $r' \rightarrow s$ for all revisions $s \in \mathit{Rid}$ such that $r \rightarrow_l^* s$; (2) for a join transition, add $r' \rightarrow_l r$.

It then remains to show that $\rightarrow_l$ is cycle-free, which we do by induction. For the initial state, this is true because $\rightarrow_l$ is empty. For other states, we need to consider the last transition, and show that it does not introduce a $\rightarrow_l$-cycle. For a fork transition, observe that the revision identifier r' is fresh, so no cycles can be formed. For a join transition, we need a bit more work, in the form of an extra Lemma 19 as stated below. With this lemma, we easily see that adding an edge $r' \rightarrow_l r$ does not introduce a cycle because it is already the case that $r' \rightarrow_l^* r$.

LEMMA 19. *Let $(s, (V, E), \rho, \gamma, o)$ be a reachable state, let $a, b \in \mathit{Rid}$ be revision identifiers, let $\rightarrow_l$ be defined as above, and let $b \in s(a)$ (that is, the revision identifier b appears in the local state of a). Then $a \rightarrow_l^* b$.*

The proof is by induction. For the initial state the claim is true because we require the initial expression to not contain any revision identifiers. Otherwise, let

$$(s, (V, E), \rho, \gamma, o) \rightarrow_d (s', (V', E'), \rho', \gamma', o')$$

be the last transition. We have to show that after the transition, $b \in s'(a)$ implies $a \rightarrow_l^* b$. We distinguish the following cases.

- fork with $a = r$ and $b = r'$. Then an edge $a \rightarrow_l b$ is added by this transition.
- fork with $a = r$ and $b \neq r'$. Then $b \in s'(a)$ implies $b \in s(a)$ and thus $a \rightarrow_l b$ by induction.
- fork with $a = r'$. Then $b \in s'(a)$ implies $b \in s(r)$ and thus $r \rightarrow_l b$ by induction, thus an edge $a \rightarrow_l b$ is added by this transition.
- join with $a = r'$. Impossible because $s'(a) = \bot$.
- join with $a = r$ and $b \in \tau'$. Then $b \in s(r')$ and thus $r' \rightarrow_l b$ by induction. Because this transition adds an edge $a \rightarrow_l r'$ the claim follows.
- join with $a = r$ and $b \notin \tau'$. Then $b \in s'(a)$ implies $b \in s(a)$, thus by induction $r \rightarrow_l a$.
- local rules, or fork/join with $a \notin \{r, r'\}$. Then $b \in s'(a)$ implies $b \in s(a)$ and the claim follows by induction.

A.4 Proof of Lemma 16

Let $(s, (V, E), \rho, \gamma, o) \rightarrow_d (s', (V', E'), \rho', \gamma', o')$ be the transition that adds y to the graph. Because of the $\xrightarrow{j}$ edge it must be an instance of the join transition (see Fig. 9) with $w = y$, and such that $x \xrightarrow{f}\xrightarrow{s}^* \gamma(r')$. Since $r' \in s(r)$ (because r' appears in the join expression) we know by Lemma 21 (stated below) that $o(r') \xrightarrow{s}\rightarrow^* \gamma(r)$. Furthermore, since $x \xrightarrow{f}\xrightarrow{s}^* \gamma(r')$ we know by Lemma 20 (stated below) that $x = o(r')$, and since $\gamma(r) \xrightarrow{s} y$, we know $x \xrightarrow{s}\rightarrow^*\xrightarrow{s} y$ as claimed.

To finish the proof of Lemma 16, we need to discharge the following two lemmas.

LEMMA 20. *Let $(s, (V, E), \rho, \gamma, o)$ be a reachable state, and let $x, y \in V$. Then, if $x \xrightarrow{f} (\xrightarrow{s}^*)\, y$ and $y = \gamma(r)$ for some r, then $x = o(r)$.*

PROOF. By induction. For the initial state, the claim is vacuously true. Otherwise, let

$$(s, (V, E), \rho, \gamma, o) \rightarrow_d (s', (V', E'), \rho', \gamma', o')$$

be the last transition, and let $x, y \in V'$ and $a \in \mathit{Rid}$ such that $x \xrightarrow{f}\xrightarrow{s}^* y$ and $y = \gamma'(a)$. We have to show $o'(a) = x$. To do so, consider all matching instantiations of the rules in Fig. 9. We distinguish three cases.

- Any rule with $y \notin \{v, w\}$. Then $a \neq r$ and $a \neq r'$ and thus $y = \gamma'(a) = \gamma(a)$ and $o'(r) = o(r)$. By induction we know $x = o'(r)$ and the claim follows.
- Any rule with $y = v$. Since $\gamma'(r) = v = y = \gamma'(a)$ we know $a = r$. Because $\gamma(r) \xrightarrow{s} y$ is the only edge into y and $x \xrightarrow{f} (\xrightarrow{s}^*)\, y$ we know $u \xrightarrow{f} (\xrightarrow{s}^*)\gamma(r)$; so by induction we know $u = o(r)$, thus also $u = o'(r) = o'(a)$ as claimed.
- Fork rule with $y = w$. Since $\gamma(r) \xrightarrow{f} y$ we must have $\gamma(r) = x$. Furthermore, $y = \gamma'(a)$ implies $a = r'$ and thus $o'(a) = \gamma(r) = x$ as claimed.

□

LEMMA 21. *Let $(s, (V, E), \rho, \gamma, o)$ be a reachable state, let $a, b \in \mathit{dom}(\gamma)$ be active revisions, and let $b \in s(a)$ (that* *is, the revision identifier b appears in the local state of a). Then $o(b) \xrightarrow{s}\to^* \gamma(a)$.*

Proof. By induction. For the initial state, the claim is vacuously true because the initial local state may not contain any revision identifiers. Otherwise, let

$$(s, (V, E), \rho, \gamma, o) \to_d (s', (V', E'), \rho', \gamma', o')$$

be the last transition, and let $a, b \in \mathsf{dom}(\gamma')$ and $b \in s'(a)$. We have to show $o'(b) \xrightarrow{s}\to^* \gamma'(a)$. To do so, consider all matching instantiations of the rules in Fig. 9. We distinguish the following cases.

- local rules with $a \neq r$, or fork or join with $b \neq r$ and $a \notin \{r, r'\}$. Then $b \in s'(a)$ trivially implies $b \in s(a)$ because $s(a) = s'(a)$, and $a, b \in \mathsf{dom}(\gamma)$ because the only element potentially in $\mathsf{dom}(\gamma') \setminus \mathsf{dom}(\gamma)$ is r' which is not equal to either a or b. By induction $o(b) \xrightarrow{s}\to^* \gamma(a)$, from which the claim follows because $o(b) = o'(b)$ and $\gamma(a) = \gamma'(a)$.
- local rules with $a = r$. Careful inspection of the evaluation rules shows that all revision ids appearing in $s'(a)$ also appear in $s(a)$. Also, both $a, b \in \mathsf{dom}(\gamma)$ (none of the rules enlarge the domain of γ), so by induction we know $o(b) \xrightarrow{s}\to^* \gamma(a)$. The claim then follows because $o(b) = o'(b)$ and $\gamma(a) \xrightarrow{s} \gamma'(a)$.
- fork with $b \neq r'$ and $a = r$. Again, $b \in s'(a)$ implies $b \in s(a)$ because $b \neq r'$. Also, $a, b \in \mathsf{dom}(\gamma)$ because only r' is added to $\mathsf{dom}(\gamma)$ and $a, b \neq r'$. The argument completes by induction as in the previous case.
- fork with $b \neq r'$ and $a = r'$. Then $b \in s'(a)$ implies $b \in s(r)$. Also, $r, b \in \mathsf{dom}(\gamma)$ because only r' is added to $\mathsf{dom}(\gamma)$ and $r, b \neq r'$. By induction we know $o(b) \xrightarrow{s}\to^* \gamma(r)$. The claim follows because $o'(b) = o(b)$ and $\gamma(r) \xrightarrow{f} \gamma'(a)$.
- fork or join with $b = r'$. This case is impossible; the only rules using r' are fork and join; for fork, $r' \notin s$ implies $r' \notin s'$, and for join, $r' \notin s'$ because it is explicitly removed.
- join with $b \neq r'$ and $a = r$. Then $b \in s'(a)$ implies either $b \in s(a)$ or $b \in s(r')$. In either case, we can complete the induction as in the previous cases (appending a $\xrightarrow{s}$ or $\xrightarrow{j}$, respectively).
- join with $b \neq r'$ and $a \neq r$. Note that neither $a = r'$ nor $b = r'$ because $\gamma'(r') = \bot$. Thus $s(a) = s'(a)$, and also $a, b \in \mathsf{dom}(\gamma)$, thus by induction $o(b) \xrightarrow{s}\to^* \gamma(a)$, from which the claim follows because $o(b) = o'(b)$ and $\gamma(a) = \gamma'(a)$.

□

A.5 Proof of Lemma 18

Assume the claim is false and pick a counterexample such that $o(r')$ is minimal. By construction we know $o(r') \xrightarrow{f}\xrightarrow{s}{}^* \gamma(r')$; let p_1 be a corresponding path. By Lemma 21 we know $o(r') \xrightarrow{s}\to^* \gamma(r)$; let p_2 be a corresponding path. By our assumption there exists a common ancestor $w \in V$ of $\{\gamma(r), \gamma(r')\}$ such that w is not an ancestor of $o(r')$. Without loss of generality, assume w is maximal, and consider two normal paths p_3 (from w to $\gamma(r)$) and p_4 (from w to $\gamma(r')$). Consider the first common vertex of p_1 and p_4 (which can not be the first vertex of p_1); the edge preceding this vertex on p_4 must be a j-edge (because the two incoming edges must be s and j, and the other incoming edge can only be an s-edge). Because p_4 is direct this implies $w \overset{j}{\twoheadrightarrow} \gamma(r')$. Since w is maximal, p_3 and p_4 must start with different edges, one being a f-edge and the other being a s-edge (Lemma 12). Since p_4 can not contain an f-edge p_3 starts with a f-edge, and because it is direct, this implies $w \overset{f}{\twoheadrightarrow} \gamma(r)$. But now, from $w \overset{j}{\twoheadrightarrow} \gamma(r')$ and $w \overset{f}{\twoheadrightarrow} \gamma(r)$, we get $w \overset{j}{\twoheadrightarrow} v$ and $w \overset{f}{\twoheadrightarrow} v$ in (V', E'), which contradicts Thm. 13 because it would imply a cycle $\rho'(w) < \rho'(v) < \rho'(w)$ in the layout order.