1 Introduction

The goal of writing parallel programs is to increase throughput by means of concurrent execution while maintaining consistency of shared data. In transactional methods, consistency is maintained by the appropriate use of synchronization primitives, such as locks. However, such programs admit multiple different interleavings of statements, resulting in nondeterministic executions.

Burckhardt, Baldassin, and Leijen present an alternative approach, namely Revisions, to maintaining consistency of shared data, in which explicit support is provided for the programmer to specify how conflicting writes to shared data must be resolved. The major advantage of this approach is that the use of revisions does not create nondeterminism.

2 Transactions vs. Revisions

Let us consider three different variants of a very simple program.

```
void foo() {
   if (y=0)
      x=1;
}
```

```
void bar() {
   if (x=0)
      y=1;
}
```

1. Sequential Execution

```
int x = 0;
int y = 0;
task t = fork { foo(); }
bar();
join t;
```

Possible Outputs

x	y
0	1
1	0
1	1

When executing sequentially, both threads operate on shared state, and all interleavings of statement are possible. As a result, there are three possible output states (as in the table above), only the first two of which are serializable

2. Transactional Execution

```
int x = 0;
int y = 0;
task t = fork { atomic { foo(); } }
atomic { bar(); }
join t;
```

Possible Outputs

x	y
0	1
1	0

Using transactional semantics, the program execution will be equivalent to *some* serializable schedule. The possible outputs are $x = 0, y = 1$ and $x = 1, y = 0$ depending on whether foo(); or bar(); was executed first.

3. Revisioned Execution

```
versioned⟨int⟩ x = 0;
versioned⟨int⟩ y = 0;
revision r = rfork { foo(); }
bar();
rjoin r;
```

Possible Outputs

x	y
1	1

Using Revision semantics, each forked revision gets its own copy of the shared state. When two revisions are joined, the states of each are merged on a per-variable basis, resulting in the output of $x = 1, y = 1$.

3 Revision Diagram

Revision diagrams aid in visualizing the flow of control in a revisioned program. The following is a revision diagram for the example above.

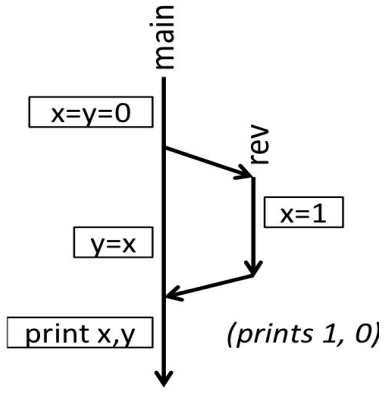
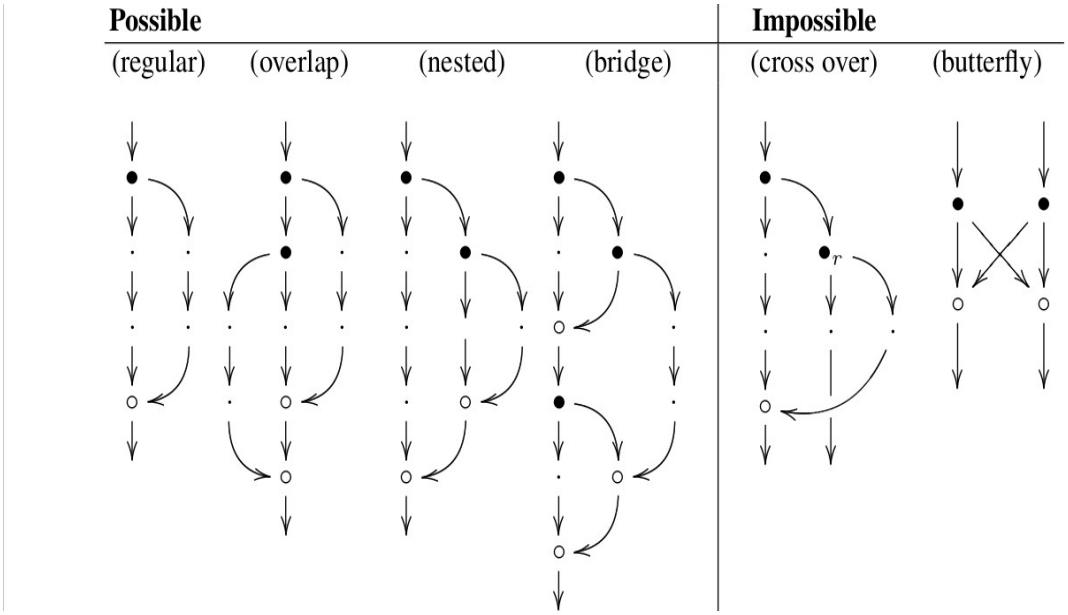


Figure 1: A revision diagram

3.1 Nested Revisions

The statement `rfork e` returns a handle to the new revision that is forked. Any revision that has access to this handle can join with the corresponding revision, enabling the creation of nested revisions. Some nestings are not possible, as the only point at which a revision handle can be passed from one revision to another is at a join. The following figure shows possible and impossible nestings.



4 Isolation Types

Isolation types are datatypes that specify a merging function to reconcile updates to a shared variable across revisions. The two isolation types defined in [1] are versioned types (`versioned`$\langle t \rangle$) and cumulative types (`cumulative`$\langle t, f \rangle$).

4.1 Versioned Types

During a join, a variable of a versioned type behaves the same as a shared variable in the absence of conflict. If there is a conflict, i.e. if both revisions udpdate a shared variable, then the value of the variable after the join is that of the joining revision.

4.2 Cumulative Types

Cumulative Types have a defined $merge$ function that specifies the value of a cumulative variable following a join. The $merge$ function takes the original value, the joined revision's value, and the joining revision's value, and produces the value resulting from the join. Here are some examples of cumulative types

1. `cumulative`$\langle int, \lambda(o, m, r).m + (r - o) \rangle$: This represents an integer variable, updates to which are additive.

2. `cumulative`$\langle int, \lambda(o, m, r).$if $r = o$ then m else $r \rangle$: This represents a `versioned`$\langle int \rangle$ as described above.

5 Datatype Granularity

Revisions do not enable maintenance of correctness constraints that involve two or more variables. For example, consider two `versioned`$\langle int \rangle$'s, x and y, that represent co-ordinates of a point. The point must satisfy the constraint that it lies within the unit circle. The program from section 2 with the corresponding revision diagram in Figure 1, invalidates the constraint although neither revision violates the constraint on

its own. The proposed solution to this problem is to define a versioned type containing both variables, for example, $\texttt{versioned}\langle\texttt{pair}\langle int\rangle\rangle$.

6 Sequential Types

It is dificult to define $merge$ functions for complex datatypes. For instance, what is a good $merge$ function for $\texttt{list}\langle int\rangle$? To address this, sequential datatypes are introduced in [2].

Definition 1. A sequential datatype is a six-tuple (S, R, M, I, ρ, μ) where

- S is a set of states
- R is a set of read operations
- M is a set of modify operations
- $I \in S$ is the initial state
- $\rho \in R \times S \to \texttt{Val}$ is the read function
- $\mu \in M \times S \to S$ is the modify function

For convenience, assume $\exists \epsilon \in M.\ \mu(\epsilon, s) = s$ for all $s \in S$.
For a sequence of operations $m_1 \cdot \ldots \cdot m_k \in M^*$, we write $\mu(m_1 \cdot \ldots \cdot m_k, s)$ to stand for $\mu(m_k, \ldots, \mu(m_1, s)\ldots)$

6.1 Examples of sequential datatypes

1. $\texttt{IntReg} = (\mathbb{Z}, \{get\}, \{set(i) \mid i \in \mathbb{Z}\}, 0, \rho, \mu)$
   where $\rho(get, i) = i$
   and $\mu(set(i'), i) = i'$

2. $\texttt{IntRegAdd} = (\mathbb{Z}, \{get\}, \{set(i), add(i) \mid i \in \mathbb{Z}\}, 0, \rho, \mu)$
   where $\rho(get, i) = i$
   and $\mu(set(i'), i) = i'$
   and $\mu(add(i'), i) = i + i'$

Definition 2. Two operations w_1, w_2 are equivalent for a sequential datatype D, written $w_1 \cong_D w_2$ if and only if

$$\forall s \in S.\ \mu(w_1, s) = \mu(w_2, s)$$

Example: $set(i) \cong_{\texttt{IntRegAdd}} set(0) \cdot add(i)$

7 Compensations

Intuitively, compensations are functions that compensate for modifications done by the other revision in the join.

Definition 3. A compensation specification for a sequential datatype D is a function

$$c^* \in (M^* \times M^*) \to (M^* \times M^*)$$

For convenience, we define

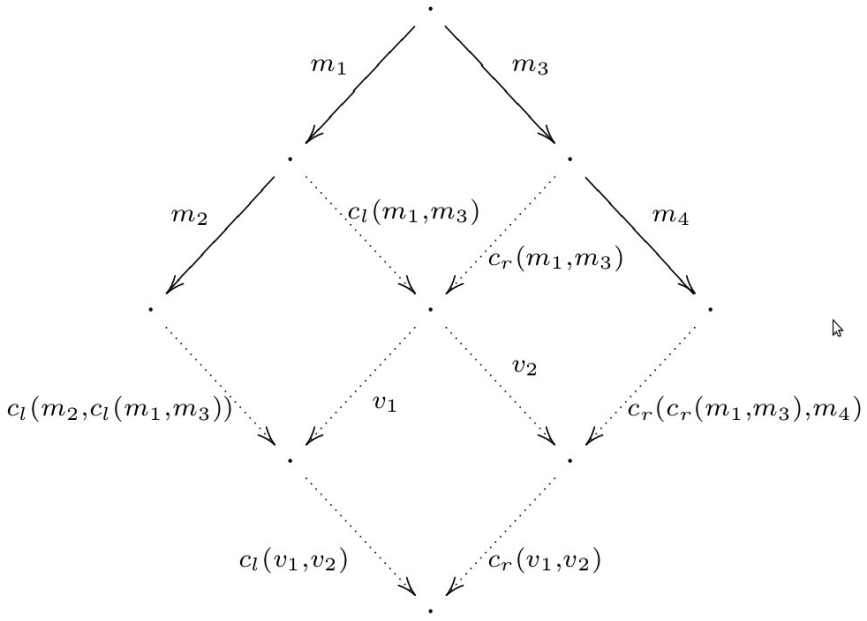
$$c_l^* \triangleq \pi_1 \circ c^*$$
$$c_r^* \triangleq \pi_2 \circ c^*$$

Definition 4. A compensation specification c^* is consistent iff

$$\forall w_l, w_m \in M^*.\ w_l \cdot c_l^*(w_l, w_r) \cong_D w_r \cdot c_r^*(w_l, w_r)$$

7.1 Compensation Tables and Tiling

A compensation table is a function $c : (M \times M) \to (M \times M)$. A compensation table uniquely defines a consisent compensation specification. The figure below shows how $c^*(m_1 \cdot m_2, m_3 \cdot m_4)$ can be computed by tiling the correspoding compensation table c.



$$v_1 = c_r(m_2, c_l(m_1, m_3))$$
$$v_2 = c_l(c_r(m_1, m_3), m_4)$$

8 Concrete Implementations of Sequential Datatypes

A concrete implementation is a tuple $(S, R, M, I, \rho, \mu, r, f)$

where	(S, R, M, I, ρ, μ)	is a sequential datatype
and	$r \in S \to S$	is a replication function
and	$f \in S \times S \times S \to S$	is a merge function

A replication function produces a new state when a fork occurs, and the merge function is isomorphic to that of cumulative types (the order of parameters is different).

8.1 Concrete Implementation of `IntRegAdd`

The compensation table `IntRegAdd` and the corresponding concrete implementation is given below. $R(i)$ represents a value that is relative, and $A(i)$, one that is absolute. $X(i)$ stands for either $R(i)$ or $A(i)$.

$$\begin{aligned}
c(add(i), add(j)) &= (add(j), add(i)) \\
c(set(i), add(j)) &= (add(j), set(i+j)) \\
c(add(i), set(j)) &= (set(j), \epsilon) \\
c(set(i), set(j)) &= (set(j), \epsilon)
\end{aligned}$$

Compensation Table for `IntRegAdd`

$$
\begin{aligned}
S &= \{A(i), R(i) \mid i \in \mathbb{Z}\} \\
I &= A(0) \\
R &= \{get\} \\
M &= \{set(i), add(i) \mid i \in Z\} \\
\rho(get, X(i)) &= i \\
\mu(set(i), X(j)) &= A(i) \\
\mu(add(i), X(j)) &= X(j+i) \\
r(X(i)) &= R(i) \\
f(\_, A(m), \_) &= A(m) \\
f(A(n), R(m), X(i)) &= A(n+m-i) \\
f(R(n), R(m), X(i)) &= X(n+m-i)
\end{aligned}
$$

Concrete Implementation of `IntRegAdd`

Acknowledgement: The figures in these notes were taken from the references.

References

[1] Sebastian Burckhardt, Alexandro Baldassin, and Daan Leijen. Concurrent programming with revisions and isolation types. In *Proceedings of the ACM international conference on Object oriented programming systems and applications (OOPSLA '10).*

[2] Sebastian Burckhardt, Manuel Fahndrich, and Daan Leijen. Roll forward, not back - A case for deterministic conflict resolution. In *The 2nd Workshop on Determinism and Correctness in Parallel Programming, 2011.*

[3] Sebastian Burckhardt and Daan Leijen. Semantics of concurrent revisions. In *Proceedings of the 20th European conference on Programming languages and systems: part of the joint European conferences on theory and practice of software (ESOP '11/ETAPS '11)*