Roll Forward, Not Back
A Case for Deterministic Conflict Resolution

Sebastian Burckhardt, Manuel Fähndrich, and Daan Leijen
Microsoft Research
{sburckha,maf,daan}@microsoft.com

Abstract

Enabling applications to execute various tasks in parallel is difficult if those tasks exhibit read and write conflicts. In recent work, we developed a programming model based on concurrent revisions that addresses this challenge: each forked task gets a conceptual copy of all locations that are declared to be shared. Each such location has a specific isolation type; on joins, state changes to each location are merged deterministically based on its isolation type. In this paper, we study how to specify isolation types abstractly using operation-based compensation functions rather than state-based merge functions. Using several examples including a list with insert, delete and modify operations, we propose compensation tables as a concise, general and intuitively accessible mechanism for determining how to merge arbitrary operation sequences. Finally, we provide sufficient conditions to verify that a state-based merge function correctly implements a compensation table.

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. However, dealing with conflicts between parallel tasks can be quite challenging with traditional synchronization models. In fact, many programmers are deterred by the engineering complexity of performing explicit, manual synchronization or replication.

Our vision is that programmers instead use the programming model of concurrent revisions [2], which simplifies parallelization of conflicting tasks by (conceptually) copying shared state automatically on a fork, and merging changes back at joins using custom merge functions. What is exciting about this model is the potential to simplify programming by using isolation types, shared higher-level data types that have suitable merge functions defined for them. For example, instead of sharing an integer to count events using read/write pair, two concurrent tasks would share a counter abstraction using increment operations on the counter instead. Although this seems like a trivial shift in perspective, it is the higher-level semantics of an increment as opposed to a read/write pair that permits the definition of “sensible” merge functions and reasoning about their behavior.

We found that even simple data types expose subtle correctness issues when trying to specify and verify them. In this paper we study a simple integer with both add and set operations in detail. We also provide merge specifications for lists with insert, modify, and delete operations. The work presented here lays the foundation for dealing with more complicated types such as maps.

In our previous work we have shown that as long as merge functions are deterministic, the entire execution model of concurrent revisions is deterministic. One particular question left unaddressed, however, is what additional properties merge functions should satisfy in order to be “sensible” for particular data types. For example, what is a sensible merge for a list data type supporting inserts, deletes, and changes to a list? We address this question in this paper by introducing compensation functions as an abstract specification mechanism. Unlike state-based merge functions, compensation functions are defined in terms of sequences of operations of the underlying data type. We make the following contributions:

1. We introduce an operation-based view of merge functions, based on compensation functions that resolve two conflicting operation sequences by appending compensations.
2. We propose compensation tables as a concise yet transparent way to specify compensation functions pairwise.
3. We show that compensation tables naturally define how to resolve arbitrary operation sequences by “tiling”.
4. We give sufficient conditions for verifying that a state-based merge function satisfies a compensation table.
5. We present compensation tables for a number of example data types, including a list, and a concrete implementation along with a detailed proof of correctness.

2. Concurrent Revisions

The context for our work is the recently proposed deterministic concurrent programming model called concurrent revisions [2,3]. Its key design principles are:
Explicit Join. The programmer forks and joins revisions, which can execute concurrently. All revisions must be joined explicitly.

Declarative Data Sharing. The programmer uses special isolation types to declare what data may be shared, and how individual data should be merged.

Effect Isolation. All changes made to shared data within a revision are only locally visible until that revision is joined.

Conceptually, the runtime copies all shared data when a new revision is forked. Therefore, the runtime can schedule concurrent revisions for parallel execution without creating data races. At the time of the join, the runtime calls a merge function for each location that was modified by the joined revision, and assigns the computed value to the location (locations that were not modified retain their current value). The function called depends on the isolation type. It is called for the versioned integers in Fig. 2. We found these versioned data types to be very useful in practice [2] as they allows us to depict concurrent computations: unlike DAGs, they are semilattices [3], and unlike in SP-graphs, children may be joined after their parent is joined (Fig. 2(e)).

3. Data Types and Compensations

We now consider some fundamental definitions of sequential data types, and show how to use compensation operations to generalize sequential semantics to a concurrent semantics appropriate for use with the concurrent revisions model.

First, let Val be the universe of values. We consider values of all types to be part of this set, and the type to be implicitly and uniquely determined by each value.

Definition 1. We define a sequential data type to be a tuple of the form \((S, R, M, I, \rho, \mu)\) 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 an initial state, \(\rho : R \times S \rightarrow Val\) is a read function (which returns for a given read operation and state the value returned by the read operation), and \(\mu : M \times S \rightarrow S\) is a modify function (returning for a given write operation and state the updated state). For convenience, we assume that every data type always includes an empty operation \(\epsilon\) such that \(\mu(\epsilon, s) = s\).

Example 2. We can define an integer register (a location holding an integer value and supporting read and write operations) as a sequential data type

\[
\text{IntReg} = (\mathbb{Z}, \{\text{get}\}, \{\text{set}(k) \mid k \in \mathbb{Z}\}, \rho, \mu)
\]

where \(\rho(\text{get}, k) = k\) and \(\mu(\text{set}(k), k') = k\).

Note that the state of a sequential data type is completely determined by the sequence of modifications. For a sequence of modifications \(w = w_1 \ldots w_n \in M^*\) and a state \(s \in S\), we write \(\mu(w, s)\) short for \(\mu(w_n, \ldots, \mu(w_1, s))\).

We consider operation sequences equivalent that are equivalent state transformers: we write \(w_1 \equiv_D w_2\) for some data type \(D = (S, R, M, I, \rho, \mu)\) if \(\mu(w_1, s) = \mu(w_2, s)\) for all \(s \in S\). For example, \(\text{set}(1) \equiv_{\text{IntReg}} \text{set}(0) \text{add}(1)\).
3.1 Constructing Merge Functions

The intention behind the concurrent revisions programming model is to behave as if all modifications performed by a revision are isolated while the revision is still running, but take effect atomically at the moment it is joined (i.e. the modifications are applied to the current state of the joining revision). As we have demonstrated in prior work [2] and in the examples earlier in this paper, we can usually achieve this with a state merge function that follows that specification? We give general answers to these questions in the remainder of this paper, with specific solutions for this Example.

3.2 Compensation Functions

It is not always clear how to devise state-based merge functions that achieve the intended semantics. Reasoning about operations can often provide more insight.

We could describe an operation-based merge function as taking two sequences of modify operations and returning a new sequence, i.e. \( f: S \times S \times S \rightarrow S \). Unfortunately, such arbitrary sequences do not permit compositional reasoning about merge behavior on nested revision diagrams (such as the one in Fig. 2(e)). Thus we use compensation specification instead.

Definition 4. A compensation specification \( c^* \) for a sequential data type \( S, R, M, I, \rho, \mu \) is a function that, given two sequences, returns two compensation sequences:

\[
\rho(M^* \times M^*) = (\rho(M^*), \rho(M^*))
\]

We write \( \rho^L, \rho^R \) to denote the left and right components of \( \rho \), respectively.

Given two operation sequences that we wish to merge, say \( w_l (\text{happening “on the left”, i.e. in the joining revision}) \) and \( w_r (\text{happening “on the right”, i.e. in the joined revision}) \), we consult \( c^* \) to obtain the compensating operation sequences, say \( c^*(w_l, w_r) = (v_l, v_r) \). The meaning of these
compute the compensating actions for merging just sequences of two operations each. Starting at the top, we first efficient implementations.

While elegant, the tiling method may however not be efficient in practice (spending time quadratic in the number of operations). This is because the effect of the merge must be equivalent to both (1) applying the compensating operations on the left, i.e. applying \( v_l \) after \( m_l \), and (2) applying the compensating operations on the right, i.e. applying \( v_r \) after \( m_r \).

**Definition 5.** A compensation specification \( c^* \) is consistent if the operations composed with their compensations are equivalent: \( \forall w_l, w_r \in M^* : w_l c^*_l(w_l, w_r) \equiv_D w_r c^*_r(w_l, w_r) \).

### 3.3 Compensation Tables

To define compensation functions in practice, it is sensible to first define a compensation function for pairs of operations \( c : M \times M \rightarrow (M \times M) \). We call such a function a compensation table. Compensation tables (if consistent) have the nice property that they uniquely define a (consistent) compensation function for arbitrary finite operation sequences because we can tile pairwise compensations as illustrated in Fig. 3, where we compute the compensation function \( c(m_1m_2, m_3m_4) \) for merging two instruction sequences of two operations each. Starting at the top, we first compute the compensating actions for merging just \( m_1 \) and \( m_3 \). From that new point we can keep merging on single operations until we fill out the 4 tiles of the diamond. Clearly, this procedure easily generalizes to sequences of more than two operations, thus defining a complete compensation function. While elegant, the tiling method may however not be efficient in practice (spending time quadratic in the number of operations). We discuss in Section 4 how to build more efficient implementations.

**Example 6.** Consider an integer data type as defined in Example 3, but with \( M \) restricted to contain only add operations. We can then define the compensation table

\[
c(\text{add}(i), \text{add}(j)) = (\text{add}(j), \text{add}(i)).
\]

In this case, the compensating action is exactly the other action. It is not hard to see that the merge we define in this way is equivalent to the state-based merge function \( f_{\text{CamInt}} \) defined earlier.

In general, if the operations commute we can always construct a consistent merge specification by using exactly the other operation as the compensating action, where \( c(m_1, m_r) = (m_r, m_1) \). Due to commutativity, if follows directly that \( m_1m_r \equiv m_r m_1 \).

The next example shows that operations do not always need to commute to be mergeable, nor does the merge function need to be symmetric.

**Example 7.** Consider the integer register defined in Example 2. We can then define the compensation table as

\[
c(\text{set}(i), \text{set}(j)) = (\text{set}(j), \epsilon).
\]

This specification is consistent since \( \text{set}(i) \text{set}(j) \equiv \text{set}(j) \epsilon \).

In this case, we want the write on the right to overwrite the write on the left, thus the compensating action on the left is \( \text{set}(j) \), while the compensating action on the right is \( \epsilon \) (none needed). Again, the merge we defined in this way is equivalent to the state-based merge function \( f_{\text{VersionedInt}} \) defined earlier.

**Example 8.** Consider the integer data type from Example 3. Then we can give a compensation table:

\[
\begin{align*}
c(\text{add}(i), \text{add}(j)) & = (\text{add}(j), \text{add}(i)) \\
c(\text{set}(i), \text{add}(j)) & = (\text{add}(j), \text{set}(i + j)) \\
c(\text{add}(i), \text{set}(j)) & = (\text{set}(j), \epsilon) \\
c(\text{set}(i), \text{set}(j)) & = (\text{set}(j), \epsilon)
\end{align*}
\]

**Example 9.** Consider a list data type \( (S, R, M, I, \rho, \mu) \) with \( S \) being the set of lists of some type (left unspecified for now), \( I \) being the empty list, \( R = \{\text{get}(i) \mid i \in \mathbb{Z}\} \), \( M = \{\text{set}(i, x) \mid i, x \in \mathbb{Z}\} \cup \{\text{ins}(i, x) \mid i, x \in \mathbb{Z}\} \cup \{\text{del}(i) \mid i \in \mathbb{Z}\} \). The insertion \( \text{ins}(i, x) \) inserts \( x \) right before the element at index \( i \). We write \( \text{idz}(m) \) to get the index from an operation, and \( \text{adj}(m, j) \) to add \( j \) to the index of an operation \( m \).

Since there are so many cases to consider, we define the merge table first just for the right-side compensation. First, we consider all pairs of operations where the indices are equal and thus work on the same element:

![Figure 3. Tiling compensation tables on single operations constructs a compensation function on sequences. In the above diagram, we have \( v_1 = c_l(m_2, c_l(m_1, m_3)) \) and \( v_2 = c_l(c_r(m_1, m_3), m_4) \).](image)
As soon as we present a concrete implementation, we would like to know whether it correctly represents the intended sequential semantics (specified by some sequential data type) and the intended merge semantics (specified by a compensation table). This question is not academic, but very important in practice; without a clear idea on how to relate the implementation to the specification, humans are certain to make mistakes. We now elaborate how we can break the verification of some concrete implementation $(S, R, M, I, \rho, \mu, r, f)$ into three conditions and give sufficient subconditions for each.

(Condition 1) To show that the implementation generalizes a sequential data type $(S', R, M', I', \rho', \mu')$ we can give an abstraction function $\psi : S \rightarrow S'$ that satisfies the following conditions:

$$\forall r \in R : \forall s \in S : \rho(r, s) = \rho'(r, \psi(s))$$
$$\forall m \in M : \forall s \in S : \psi(\mu(m, s)) = \mu'(m, \psi(s))$$
$$\psi(I) = I'$$
$$\forall s \in S : \psi(r(s)) = \psi(s)$$

**Example 12.** We can show that the implementation in Example 11 generalizes the sequential data type Int from Example 3 by defining the map $\psi : S \rightarrow \mathbb{Z}$ as $\psi(X(k)) = k$, that is, to “erase” the extra information. The cases are then easily verified.

(Condition 2) We can show that the concrete implementation correctly merges revisions in cases where there is at most one operation, by enumerating all cases. Specifically, for all $s \in S$ and $m_1, m_2 \in M$, we can show:

$$f(\mu(m_1, s), \mu(m_2, r(s)), s) = \mu(m_1 \cdot \mu(m_2, m_1), s) = \mu(m_2 \cdot \mu(m_2, m_1), s)$$

**Example 13.** For the implementation in Example 11 and the compensation table in Example 8, we can discharge this condition by going through all the cases for $m_1$ and $m_2$, each case being relatively simple.

(Condition 3) We can show that the concrete implementation merges states correctly even if multiple operations need to be reconciled, by showing that there exists a “smash” function $\xi : M \times M \rightarrow M$ that satisfies the following conditions:
1. $\xi$ is associative.
2. $\forall m \in M: \xi(m, e) = \xi(e, m) = m$.
3. $\xi$ is consistent with $\mu$: for all $s \in S$ and $m_1, m_2 \in M$, we have $\mu(\xi(m_1, m_2), s) = \mu(m_2, \mu(m_1, s))$.
4. $\xi$ is consistent with tiling of compensation functions (as in Fig. 3): for all $m_1, m_3, m_4 \in M$, we have
   \[ c_1 \left( m_1, m_3, m_4 \right) = \xi(c_1 \left( m_1, m_3 \right), c_1 \left( c_1 \left( m_1, m_3 \right), m_4 \right) \right) \]
   \[ c_r \left( m_1, m_3, m_4 \right) = c_r \left( c_r \left( m_1, m_3, m_4 \right) \right) \]
   and for all $m_1, m_2, m_3 \in M$, we have
   \[ c_1 \left( m_1, m_2, m_3 \right) = c_1 \left( m_2, c_l \left( m_1, m_3 \right) \right) \]
   \[ c_r \left( m_1, m_2, m_3 \right) = \xi(c_r \left( m_1, m_3, m_4 \right), c_r \left( m_2, c_l \left( m_1, m_3 \right) \right) \right) \]

**Example 14.** For the implementation in Example 11 and the compensation table in Example 8, we define the smash function as follows:

\[ \xi(\text{add}(i), \text{add}(j)) = \text{add}(i + j) \]
\[ \xi(\text{add}(i), \text{set}(j)) = \text{set}(j) \]
\[ \xi(\text{set}(i), \text{add}(j)) = \text{set}(i + j) \]
\[ \xi(\text{set}(i), \text{set}(j)) = \text{set}(j) \]

Again, we can then discharge the conditions by going through all the cases. The first three are easy. Consistency with the compensation functions is abit more work. Listing all 32 cases appeared overwhelming at first, but using diagrams simplified the task reasonably; we drew and filled in one diagram for each of the 8 combinations of $m_1, m_3, m_4$, and one diagram for each of the 8 combinations of $m_1, m_2, m_3$, then checked 2 conditions per diagram. Clearly, for more complex data types we would automate this process.

5. **Related Work**

Recently, researchers have proposed programming models for deterministic concurrency [1, 5, 11, 14]. These models all guarantee that the execution is equivalent to some sequential execution and do not resolve true conflicts. Cilk++ hyperobjects [6] are similar to isolation types, but are deterministic only for fully commutative operations, and Cilk tasks follow a more restricted concurrency model [7, 12]. Isolation types are also similar to the idea of coarse-grained transactions [8] and semantic commutativity [9] insofar they eliminate false conflicts by raising the abstraction level.

Conflict resolution schemes have also been studied in the context of collaborative editing and eventual consistency. Most similar to our compensation function idea is the operational transformations approach [4], but it requires more complicated consistency conditions often violated by actual implementations [10]. Alternatively, conflict resolution can be simplified by making all operations commutative [13].

6. **Conclusion**

We believe that the concurrent revisions framework is a great foundation to study mergeable datatypes. Using compensation tables we can concisely specify the semantics of concurrent data types, and we are working to specify more complex data types like graphs and dictionaries.

**References**


