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 and writes, 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 unadressed, 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 specfication 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 a 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 f 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 with three values $v_{\text{current}}, v_{\text{joined}}$ and v_{original} representing the current value in the joining revision, the current value in the joined revision, and the value at the time the revision was originally forked, respectively.

For example, a cumulative integer type may employ the merge function $f_{\textit{CumulativeInt}}(v_c, v_j, v_o) = v_c + (v_j - v_o)$ as shown in Fig. 1. This function computes the relative effect $v_j - v_o$ of the modifications in the joined revision and adds it to the current value v_c. The effect is that modifications by all revisions are cumulative.

Another isolation type may give priority to the writes in the child revision by specifying $f(v_c, v_j, v_o) = v_j$, as we do for the versioned integers in Fig. 2. We found these versioned types to be very useful in practice [2] as they allows us to precisely control the order of writes: all writes to a versioned variable appear to take effect atomically at the time of the join (and thus simply overwrite earlier writes).

A key benefit of the "concurrent revisions + isolation types" model is that it resolves conflicts between tasks in a deterministic and programmable way. Even if tasks exhibit conflicts, they can still be efficiently executed in parallel, without issues caused by unnecessary rollbacks and retries. Moreover, the computation is determinate, meaning that it is fully specified by the program and does not depend on the relative timing or scheduling of tasks[1] [3].

2.1 Revision Diagrams

It is often helpful to visualize computations using *revision diagrams* (see Fig. 1(b) and the bottom row of Fig. 2). Such diagrams show each revision as a vertically aligned sequence of points, each point representing one (dynamic) instance of a statement. We use curved arrows to show precisely where

[1] Note further that the determinacy also does not depend on the merge function being associative or commutative (or being "sensible" in any other way, for that matter); the only requirement is that it be a function in the mathematical sense.

(a)

```
CumulativeInt x = 0 ;
r = fork { x = x + 2 ; }
x = x + 3 ;
join r ; //x = 3 + (2 − 0)
assert(x ≡ 5) ;
```

(b)

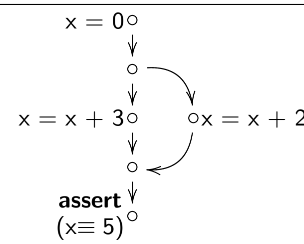

Figure 1. Example illustrating parallel aggregation with the isolation type CumulativeInt with the merge function $f_{\textit{CumulativeInt}}(v_c, v_j, v_o) = v_c + (v_j - v_o)$.

new revisions branch out (on a fork) and where they merge in (on a join). As shown in Fig. 2(e), revisions can be nested.

Revision diagrams are not equivalent to other graphs commonly used 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, 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 an initial state, $\rho : R \times S \to Val$ is a read function (which returns for a given read operation and state the value returned by the read operation), and $\mu : M \times S \to S$ 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 ϵ 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*

$$\textit{IntReg} = (\mathbb{Z}, \{\textit{get}\}, \{\textit{set}(k) \mid k \in \mathbb{Z}\}, \rho, \mu)$$

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

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

We consider operation sequences equivalent that are equivalent state transformers: we write $w_1 \cong_D w_2$ 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, $\textit{set}(1) \cong_{\textit{IntReg}} \textit{set}(0)\textit{add}(1)$.

(a)

```
versioned⟨int⟩ x = 0 ;
r = fork { x = 2 ; }
x = 1 ;
join r ;
assert(x ≡ 2) ;
```

(b)

```
versioned⟨int⟩ x = 0 ;
r = fork { x = x ; }
x = 1 ;
join t ;
assert(x ≡ 0) ;
```

(c)

```
versioned⟨int⟩ x = 0 ;
r = fork { }
x = 1 ;
join t ;
assert(x ≡ 1) ;
```

(d)

```
versioned⟨int⟩ x = 0 ;
r1 = fork { x = 2 ; }
r2 = fork { x = 3 ; }
join r2 ;
join r1 ;
assert(x ≡ 2) ;
```

(e)

```
versioned⟨int⟩ x = 0 ;
r1 = fork { r2 = fork { x = 1 ; } }
join r1 ;
assert(x ≡ 0) ;
join r2 ;
assert(x ≡ 1) ;
```

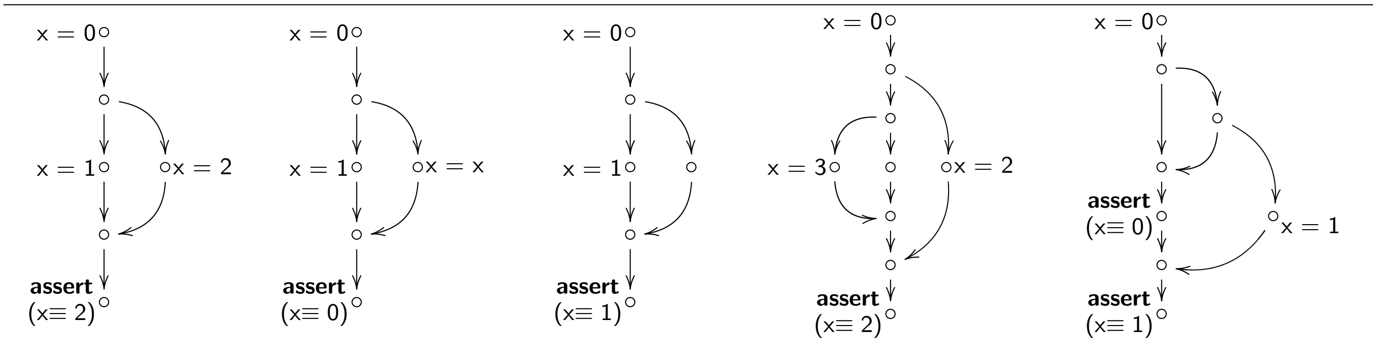

Figure 2. Examples illustrating the semantics of revisions and versioned types with the merge function $f(v_c, v_j, v_o) = v_j$. **(a)** The write x=2 takes effect on join; **(b,c)** the assignment x=x counts as a write, thus removing it leads to a different result; **(d)** the order of joins determines the order of writes; **(e)** a nested revision may be joined after its parent.

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 $f : S \times S \times S \to S$ that is invoked during join if there were write-write conflicts, i.e. if both the parent and the child performed modifications.

For example, the versioned integer type illustrated in Fig.2 is intended to merges modifications that represent *absolute* changes to the value. On the other hand, the cumulative integer type shown in Fig. 1 is intended to merge modifications that are interpreted as *relative* changes. But what if some changes are absolute and some are relative? In that case, we need more information about what operations were performed to perform a proper merge.

EXAMPLE 3. *We can define a sequential data type* Int *that offers two different modify operations, an absolute one (set it to specific value), and a relative one (add a value) as follows:*

$$
\begin{aligned}
&S = \mathbb{Z}\\
&R = \{get\}\\
&M = \{set(k) \mid k \in \mathbb{Z}\} \cup \{add(k) \mid k \in \mathbb{Z}\}\\
&\rho(get, k) = k\\
&\mu(set(k), k') = k\\
&\mu(add(k), k') = k' + k
\end{aligned}
$$

This example raises two questions: how do we specify what should happen if operations $add(k)$ and $set(k)$ are called concurrently? And how can we implement 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. $f : M^* \times M^* \to M^*$. 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, ρ, μ) *is a function that, given two sequences, returns two* compensation sequences*:*

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

We write c^*_l, c^*_r to denote the left and right components of c^*, respectively.

Given two operation sequences that we wish to merge, say w_l (happening "on the left", i.e. in the joining revision) and w_r (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

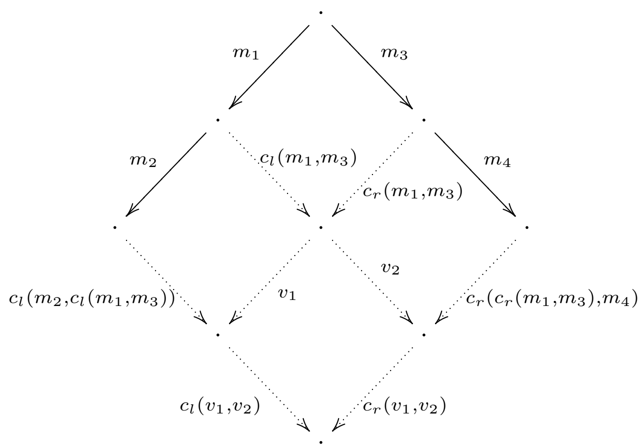

Figure 3. Tiling compensation tables on single operations constructs a compensation function on sequences. In the above diagram, we have $v_1 = c_r(m_2, c_l(m_1, m_3))$ and $v_2 = c_l(c_r(m_1, m_3), m_4)$.

operations is that 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:* $\forall w_l, w_r \in M^* : w_l c_l^*(w_l, w_r) \cong_D w_r c_r^*(w_l, w_r)$.

3.3 Compensation Tables

To define compensation functions in practice, it is sensible to first define a compensation function for pairs of operations $c : M \times M \to (M \times M)$. 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 $c(m_1 m_2, m_3 m_4)$ 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*

$$c(\mathsf{add}(i), \mathsf{add}(j)) = (\mathsf{add}(j), \mathsf{add}(i)).$$

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_{CumInt} *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 $c(m_l, m_r) = (m_r, m_l)$. Due to commutativity, if follows directlty that $m_l m_r \cong m_r m_l$.

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*

$$c(\mathsf{set}(i), \mathsf{set}(j)) = (\mathsf{set}(j), \epsilon).$$

This specification is consistent since $\mathsf{set}(i)\mathsf{set}(j) \cong \mathsf{set}(j)\epsilon$. *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* $\mathsf{set}(j)$*, while the compensating action on the right is* ϵ *(none needed). Again, the merge we defined in this way is equivalent to the state-based merge function* $f_{\mathsf{VersionedInt}}$ *defined earlier.*

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

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

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

$$\begin{array}{lll}
c_r(\textit{del}(i), \textit{del}(i)) & = \epsilon & \textit{(double deletion)} \\
c_r(\textit{set}(i,x), \textit{del}(i)) & = \epsilon & \textit{(set is deleted)} \\
c_r(\textit{set}(i,x), \textit{set}(i,y)) & = \epsilon & \textit{(right side wins)} \\
c_r(\textit{ins}(i,x), \textit{ins}(i,y)) & = \textit{ins}(i,x) & \textit{if } x < y \\
c_r(\textit{ins}(i,x), \textit{ins}(i,y)) & = \textit{ins}(i+1,x) & \textit{if } x \geq y \\
 & & \\
c_r(\textit{del}(i), \textit{set}(i,x)) & = \textit{del}(i) & \\
c_r(\textit{ins}(i,x), \textit{del}(i)) & = \textit{ins}(i,x) & \\
c_r(\textit{del}(i), \textit{ins}(i,x)) & = \textit{del}(i+1) & \\
c_r(\textit{set}(i,x), \textit{ins}(i,y)) & = \textit{set}(i+1,x) &
\end{array}$$

Secondly, we consider the cases where the indices differ:

$$\begin{array}{lll}
c_r(m_1, m_2) & = m_1 & \textit{if } idx(m_1) < idx(m_2) \\
c_r(m_1, \textit{del}(i)) & = adj(m_1, -1) & \textit{if } idx(m_1) > i \\
c_r(m_1, \textit{ins}(i,x)) & = adj(m_1, +1) & \textit{if } idx(m_1) > i \\
c_r(m_1, \textit{set}(i,x)) & = m_1 & \textit{if } idx(m_1) > i
\end{array}$$

Finally, we define the left compensation in terms of the right compensation:

$$\begin{array}{lll}
c_l(\textit{set}(i,x), \textit{set}(i,y)) & = \textit{set}(i,y) & \textit{(right side wins)} \\
c_l(m_1, m_2) & = c_r(m_2, m_1) & \textit{otherwise}
\end{array}$$

This defines the merge semantics of mutable lists. Note that the cases for conflicting set*'s prefer the* set *of the right side. One could define lists over mergeable types and use the apropiate merge specification for that type in such cases. For double insertions $c_r(\textit{ins}(i,x), \textit{ins}(i,y))$, we use the order between the elements x and y to determine the final order: this ensures that for example a sorted list stays sorted after merging. Another choice could be to use 'right side first', to only preserve the relative order of insertions in each branch.*

4. Concrete Implementations

In practice, we often try to construct concrete implementations of a data type where state forking and merging is represented by a replication function (representing the operation of copying the state upon a fork for use in the forked revision) and a state merge function f as described in Section 2.

DEFINITION 10. *We define a* concrete implementation *to be a tuple $(S, R, M, I, \rho, \mu, r, f)$ where (S, R, M, I, ρ, μ) is a sequential data type, $r : S \to S$ is a replication function, and $f : S \times S \times S \to S$ is a merge function.*

EXAMPLE 11. *We can define a concrete integer implementation for the* Int *type from Example 3 that satisfies the compensation tables given in Example 8. We augment the state to store not just a simple integer k, but $A(k)$ if the integer should be interpreted as an absolute value, or $R(k)$ if it should be interpreted as a relative value. To simplify the notation below, we use _ as a pattern that matches an arbitrary state, and $X(k)$ as a pattern that matches $A(k)$ or $R(k)$.*

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

4.1 Verifying Concrete Implementations

As soon as we present a concrete implementation, we would like to know whether it correctly represents the 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 $\psi : S \to S'$ that satisfies the following conditions:

$$\begin{array}{l}
\forall r \in R : \forall s \in S : \rho(r,s) = \rho'(r, \psi(s)) \\
\forall m \in M : \forall s \in S : \psi(\mu(m,s)) = \mu'(m, \psi(s)) \\
\psi(I) = I' \\
\forall s \in S : \psi(r(s)) = \psi(s)
\end{array}$$

EXAMPLE 12. *We can show that the implementation in Example 11 generalizes the sequential data type* Int *from Example3 by defining the map $\psi : S \to \mathbb{Z}$ 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:

$$\begin{array}{rcl}
f(\mu(m_1, s), \mu(m_2, r(s)), s) & = & \mu(m_1 c_l(m_1, m_2), s) \\
 & = & \mu(m_2 c_r(m_2, m_1), s)
\end{array}$$

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 $\xi : M \times M \to M$ that satisfies the following conditions:

1. ξ is associative.
2. $\forall m \in M : \xi(m, \epsilon) = \xi(\epsilon, m) = m$.
3. ξ is consistent with μ: 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. ξ is consistent with tiling of compensation functions (as in Fig. 3): for all $m_1, m_3, m_4 \in M$, we have

$$
\begin{aligned}
c_l(m_1, \xi(m_3, m_4)) &= \xi(c_l(m_1, m_3), c_l(c_r(m_1, m_3), m_4)) \\
c_r(m_1, \xi(m_3, m_4)) &= c_r(c_r(m_1, m_3), m_4))
\end{aligned}
$$

and for all $m_1, m_2, m_3 \in M$, we have

$$
\begin{aligned}
c_l(\xi(m_1, m_2), m_3) &= c_l(m_2, c_l(m_1, m_3))) \\
c_r(\xi(m_1, m_2), m_3) &= \xi(c_r(m_1, m_3), c_r(m_2, c_l(m_1, m_3)))
\end{aligned}
$$

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

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

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

[1] 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.

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

[3] S. Burckhardt and D. Leijen. Semantics of concurrent revisions. In *European Symposium on Programming (ESOP)*, 2011.

[4] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. *SIGMOD Rec.*, 18:399–407, June 1989.

[5] R. B. et al. A type and effect system for Deterministic Parallel Java. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2009.

[6] M. Frigo, P. Halpern, C. E. Leiserson, and S. Lewin-Berlin. Reducers and other Cilk++ hyperobjects. In *Symposium on Parallel Algorithms and Architectures (SPAA)*, 2009.

[7] 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.

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

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

[10] G. Oster, P. Urso, P. Molli, and A. Imine. Proving correctness of transformation functions in collaborative editing systems. Rapport de Recherche 5795, LORIA – INRIA Lorraine, Dec. 2005.

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

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

[13] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Rapport de Recherche 7506, INRIA, Jan. 2011.

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