A Formal Semantics for Skip's Reactive `reduce` Combinator

Cristiano Calcagno

November 2025

Abstract

We present a formal semantics for the `reduce` combinator in Skip [1], a reactive programming framework that maintains aggregated views of collections with automatic incremental updates. Skip exposes reducers as user-facing combinators: a reducer $R = (\iota, \oplus, \ominus)$ specifies an initial value, an add operation, and a remove operation, subject to an informal correctness condition. This paper formalizes that condition and proves that incremental correctness—where updates produce the same result as recomputation—holds *if and only if* the reducer is well-formed (i.e., $\ominus$ is the inverse of $\oplus$). Our contribution combines the **algebraic foundations of incremental computation** (from differential dataflow and databases) with Skip's **user-facing combinator design** to provide a **formally characterized correctness contract**—giving Skip users a precise specification for writing correct custom reducers.

1 Introduction

Skip [1] is a reactive programming framework that maintains derived views of collections with automatic incremental updates. When the underlying data changes, Skip efficiently propagates updates to all dependent computations without manual intervention. A central operation in Skip is the `reduce` combinator, which computes a summary (or *view*) for each key in a collection by folding over its associated values—for example, computing the sum, count, or minimum.

The key to efficient updates is that Skip's `reduce` supports *retractions*: when values are removed from a collection, the reducer can incrementally update the accumulated result rather than recomputing from scratch. Skip exposes this capability through a *reducer* abstraction $R = (\iota, \oplus, \ominus)$, where ι is an initial value, $\oplus$ is an add operation, and $\ominus$ is a remove operation. This allows $O(1)$ updates per change, rather than $O(n)$ recomputation.

Skip's user-facing combinators. A distinguishing feature of Skip is that reducers are exposed as **user-facing combinators**—first-class programming

constructs that developers use directly to build custom reactive services. Rather than being internal implementation details hidden from users, Skip allows developers to define their own reducers for domain-specific aggregations. Skip's documentation specifies an informal correctness condition: the result of applying the runtime's sequence of remove/add calls must equal recomputing from scratch. This design enables extensibility (custom aggregations), composability (reducers combine with other Skip combinators), and—when the condition is satisfied—correctness guarantees.

This paper. We formalize Skip's reducer abstraction and its correctness condition. Our main result (Theorem 2) proves that incremental correctness holds *if and only if* the reducer is well-formed—that is, $\ominus$ is the inverse of $\oplus$. This provides Skip users with a precise specification: satisfy the well-formedness condition, and your custom reducer is guaranteed to work correctly with Skip's incremental update mechanism.

Synthesis of ideas. Our formalization builds on Skip's design and synthesizes ideas from multiple domains:

- From **incremental databases and streaming systems**, we take the insight that invertibility is key to efficient updates—aggregations with inverse operations can be maintained in $O(1)$ time per change.
- From **Skip's reactive runtime**, we take the user-facing combinator design: a first-class, composable operator that developers use directly, rather than callbacks tied to internal machinery.
- From **formal methods**, we provide a complete characterization: not merely sufficient conditions for correctness, but a precise *if-and-only-if* theorem that fully characterizes when Skip's incremental updates are correct.

Contributions. This paper formalizes Skip's `reduce` combinator. We provide:

- A denotational semantics for `reduce` as a derived view (Section 3)
- A formal model of deltas and Skip's incremental update procedure (Section 4)
- A precise well-formedness condition and proof that it is both necessary and sufficient for correctness (Section 5)
- Concrete examples including sum, count, and min reducers (Section 6)

We also position Skip's design relative to other streaming and reactive systems (Section 8) and analyze the complexity benefits of incremental updates.

2 Preliminaries

Let K be a set of keys, V a set of values, and A a set of accumulator values. For a set V, we write $\mathcal{M}(V)$ for the set of finite multisets over V; we use $\uplus$ and $\setminus$ for multiset union and multiset difference, respectively.

Definition 1 (Collection)**.** *A* collection *is a function $C : K \to \mathcal{M}(V)$. We write $C(k)$ for the multiset of values associated with key k.*

2.1 Commutative Operations

Definition 2 (Pairwise Commutative Operation)**.** *Let $\star : A \times V \to A$ be an update operation. We say that $\star$ is* pairwise commutative *if*

$$\forall a \in A, v_1, v_2 \in V.\ (a \star v_1) \star v_2 = (a \star v_2) \star v_1.$$

2.2 Folds

Definition 3 (Fold over Sequence for an Operation)**.** *Let $\star : A \times V \to A$ be an update operation and let $s = [v_1, \ldots, v_n]$ be a finite sequence of elements of V. For any $a \in A$ we define:*

$$\mathsf{fold}^{\mathsf{seq}}_{\star}(a, []) = a \quad \textit{and} \quad \mathsf{fold}^{\mathsf{seq}}_{\star}(a, v_1 :: s') = \mathsf{fold}^{\mathsf{seq}}_{\star}(a \star v_1, s').$$

When a distinguished initial element $\iota \in A$ is understood from context, we write $\mathsf{fold}^{\mathsf{seq}}_{\star}(s)$ for $\mathsf{fold}^{\mathsf{seq}}_{\star}(\iota, s)$.

Theorem 1 (Characterisation of Multiset Fold)**.** *Let $\star : A \times V \to A$ be an update operation. The following are equivalent:*

1. *For all $a \in A$, $M \in \mathcal{M}(V)$ and any two finite sequences s_1, s_2 enumerating M (with multiplicity), we have*

   $$\mathsf{fold}^{\mathsf{seq}}_{\star}(a, s_1) = \mathsf{fold}^{\mathsf{seq}}_{\star}(a, s_2).$$

   That is, folding depends only on the multiset of elements, not on their enumeration.

2. *The operation $\star$ is pairwise commutative in the sense of Definition 2.*

Proof. Sketch: For $(2 \Rightarrow 1)$, one shows first that swapping two adjacent elements in a sequence does not change the fold, using pairwise commutativity of $\star$. Since any permutation of a finite sequence can be written as a product of adjacent transpositions, it follows that the fold depends only on the underlying multiset. For $(1 \Rightarrow 2)$, instantiate (1) with the two sequences $[v_1, v_2]$ and $[v_2, v_1]$ enumerating the same multiset $\{v_1, v_2\}$, and expand the definition of the fold to obtain $(a \star v_1) \star v_2 = (a \star v_2) \star v_1$. □

Definition 4 (Fold over Multiset for an Operation)**.** *Let $\star : A \times V \to A$ be pairwise commutative and let $M \in \mathcal{M}(V)$ be finite. For $a \in A$ and any sequence s enumerating M (with multiplicity), we set*

$$\mathsf{fold}_\star(a, M) := \mathsf{fold}^{\mathsf{seq}}_\star(a, s),$$

which is well-defined by the Characterisation of Multiset Fold. If an initial element $\iota \in A$ is fixed, we abbreviate $\mathsf{fold}_\star(M) := \mathsf{fold}_\star(\iota, M)$.

3 The Reduce Combinator

The `reduce` combinator produces a *view* of a collection by summarizing the values for each key.

Definition 5 (Reduce Combinator)**.** *Let $\oplus : A \times V \to A$ be a pairwise commutative operation and $\iota \in A$ an initial value. Given a collection $C : K \to \mathcal{M}(V)$, the* reduce *combinator produces a view:*

$$\mathsf{reduce}_{\iota,\oplus}(C) : K \to A$$

defined as:

$$\mathsf{reduce}_{\iota,\oplus}(C)(k) = \mathsf{fold}_\oplus(\iota, C(k))$$

That is, for each key k, we fold the operation $\oplus$ over all values in $C(k)$, starting from ι.

The view $\mathsf{reduce}_{\iota,\oplus}(C)$ is a derived collection that depends on C. When C changes, the view must be updated to remain consistent. The next section addresses how to perform these updates efficiently.

4 Incremental Updates

When a collection C changes, the view $\mathsf{reduce}_{\iota,\oplus}(C)$ must be updated. A naïve approach would recompute the fold from scratch for each affected key, requiring $O(n)$ time where n is the size of the multiset. To achieve $O(1)$ updates, we introduce a *remove* operation $\ominus$ that can undo the effect of $\oplus$.

4.1 Reducers

Definition 6 (Reducer)**.** *A* reducer *is a triple $R = (\iota, \oplus, \ominus)$ where $\iota \in A$ is an initial value, and*

$$\oplus, \ominus : A \times V \to A$$

are update operations such that both $\oplus$ and $\ominus$ are pairwise commutative in the sense of Definition 2. We call $\oplus$ the add *operation and $\ominus$ the* remove *operation.*

For a reducer $R = (\iota, \oplus, \ominus)$, we write reduce_R for $\mathsf{reduce}_{\iota,\oplus}$.

Definition 7 (Well-Formed Reducer)**.** *A reducer* $R = (\iota, \oplus, \ominus)$ *is* well-formed *if* $\ominus$ *is the* ***inverse*** *of* $\oplus$*:*

$$\forall a \in A, v \in V.\ (a \oplus v) \ominus v = a$$

In database terminology, a well-formed reducer defines an *invertible distributive aggregate*: the fold can be computed over partitions independently (distributive), and individual values can be removed from the accumulated result (invertible).

Remark 1 (Remove-Add Commutativity)**.** *For well-formed reducers where* $\oplus$ *and* $\ominus$ *arise from an abelian group action on* A*, the following property holds automatically:*

$$\forall a \in A, v_1, v_2 \in V.\ (a \ominus v_1) \oplus v_2 = (a \oplus v_2) \ominus v_1$$

This ensures that the order of interleaved adds and removes does not affect the final result. All practical reducers (sum, count, product over commutative groups) satisfy this.

4.2 Deltas

We model updates to collections as deltas.

Definition 8 (Delta)**.** *A* delta Δ *for a collection* C *is a pair* (Δ^+, Δ^-) *where:*

- $\Delta^+ : K \to \mathcal{M}(V)$ *represents added values*
- $\Delta^- : K \to \mathcal{M}(V)$ *represents removed values*

We require that $\Delta^-(k) \subseteq C(k)$ *for all* k *(i.e., we can only remove values that exist in the collection).*

Definition 9 (Delta Application)**.** *Given a collection* C *and a delta* $\Delta = (\Delta^+, \Delta^-)$ *for* C*, the updated collection* $C \bullet \Delta$ *is defined pointwise by:*

$$C \bullet \Delta = \lambda k.\, (C(k) \setminus \Delta^-(k)) \uplus \Delta^+(k)$$

Intuitively, $C \bullet \Delta$ *is the collection obtained by first removing all values in* Δ^- *from* C *and then adding all values in* Δ^+*.*

Remark 2 (Operational Construction of Deltas)**.** *In the Skip runtime, for each key* k *we compute an* old *multiset* $old(k)$ *of contributing values and a* new *multiset* $new(k)$*. The delta is then constructed as:*

$$\Delta^+(k) = new(k) \setminus old(k) \quad \textit{and} \quad \Delta^-(k) = old(k) \setminus new(k).$$

Note that Δ^+ *and* Δ^- *are disjoint by construction. If* C_{old} *is the collection with* $C_{old}(k) = old(k)$*, then* $\Delta^-(k) \subseteq C_{old}(k)$*, so* Δ *is a valid delta for* C_{old}*. Moreover,* $C_{old} \bullet \Delta = C_{new}$ *where* $C_{new}(k) = new(k)$*.*

4.3 Incremental Update

Definition 10 (Incremental Reduce)**.** *Given the current accumulator value a_k for key k and a delta Δ, the new accumulator is computed as:*

$$\mathsf{update}_R(a_k, \Delta, k) = \mathsf{fold}_{\oplus}\big(\mathsf{fold}_{\ominus}(a_k, \Delta^-(k)),\ \Delta^+(k)\big)$$

That is, we first apply all removals $\Delta^-(k)$ to a_k using $\ominus$, and then apply all additions $\Delta^+(k)$ using $\oplus$. This is well-defined since $\oplus$ and $\ominus$ are pairwise commutative.

5 Correctness

We now characterize exactly when incremental updates are correct. At some moment we have an *old* collection C and, for each key k, an accumulator

$$a_k = \mathsf{reduce}_R(C)(k)$$

that agrees with the denotational semantics. A change to the collection is described abstractly by a delta Δ, yielding the *updated* collection

$$C' = C \bullet \Delta.$$

For each key k there are then two ways to obtain the *new* accumulator value:

- *Denotational recompute:* ignore a_k and compute

  $$a'_k = \mathsf{reduce}_R(C')(k),$$

  i.e. start from ι and fold $\oplus$ over the current multiset $C'(k)$.

- *Incremental update:* update the old accumulator a_k using the delta by

  $$a'_k = \mathsf{update}_R(a_k, \Delta, k),$$

  i.e. first remove $\Delta^-(k)$ using $\ominus$, then add $\Delta^+(k)$ using $\oplus$.

Definition 11 (Incremental Correctness Property)**.** *A reducer R satisfies the* incremental correctness property *if for all collections C, all valid deltas Δ for C, and all keys k:*

$$\mathsf{reduce}_R(C \bullet \Delta)(k) = \mathsf{update}_R(\mathsf{reduce}_R(C)(k), \Delta, k)$$

The following theorem shows that the inverse property is both necessary and sufficient for incremental correctness.

Theorem 2 (Characterization of Incremental Correctness)**.** *Let $R = (\iota, \oplus, \ominus)$ be a reducer (with pairwise commutative $\oplus$ and $\ominus$). The following are equivalent:*

1. *R is well-formed (i.e., $(a \oplus v) \ominus v = a$ for all $a \in A$, $v \in V$).*

2. R satisfies the incremental correctness property.

Proof. We prove both directions.

(1 ⇒ 2): Well-formedness implies correctness.

Assume R is well-formed. Let C be a collection, $\Delta = (\Delta^+, \Delta^-)$ a valid delta for C, and k a key. Write $M = C(k)$ for the old multiset, $M' = C'(k) = (M \setminus \Delta^-(k)) \uplus \Delta^+(k)$ for the new multiset, and $a = \mathsf{fold}_\oplus(\iota, M)$ for the old accumulator.

We must show $\mathsf{fold}_\oplus(\iota, M') = \mathsf{fold}_\oplus(\mathsf{fold}_\ominus(a, \Delta^-(k)), \Delta^+(k))$.

Since $\Delta^-(k) \subseteq M$, we can write $M = M_0 \uplus \Delta^-(k)$ for some multiset M_0. By pairwise commutativity of $\oplus$:

$$a = \mathsf{fold}_\oplus(\iota, M) = \mathsf{fold}_\oplus(\iota, M_0 \uplus \Delta^-(k)) = \mathsf{fold}_\oplus(\mathsf{fold}_\oplus(\iota, M_0), \Delta^-(k))$$

Let $a_0 = \mathsf{fold}_\oplus(\iota, M_0)$. Then $a = \mathsf{fold}_\oplus(a_0, \Delta^-(k))$.

By the inverse property applied inductively (once for each element of $\Delta^-(k)$):

$$\mathsf{fold}_\ominus(a, \Delta^-(k)) = \mathsf{fold}_\ominus(\mathsf{fold}_\oplus(a_0, \Delta^-(k)), \Delta^-(k)) = a_0$$

Therefore:

$$\mathsf{fold}_\oplus(\mathsf{fold}_\ominus(a, \Delta^-(k)), \Delta^+(k)) = \mathsf{fold}_\oplus(a_0, \Delta^+(k))$$

Since $M' = M_0 \uplus \Delta^+(k)$:

$$\mathsf{fold}_\oplus(\iota, M') = \mathsf{fold}_\oplus(\mathsf{fold}_\oplus(\iota, M_0), \Delta^+(k)) = \mathsf{fold}_\oplus(a_0, \Delta^+(k))$$

Thus both sides are equal.

(2 ⇒ 1): Correctness implies well-formedness.

Assume R satisfies the incremental correctness property. We must show $(a \oplus v) \ominus v = a$ for all $a \in A$ and $v \in V$.

Fix $a \in A$ and $v \in V$. We first establish the property for a of the form $a = \mathsf{fold}_\oplus(\iota, M)$ for some multiset M.

Define a collection C with a single key k where $C(k) = M \uplus \{v\}$. Then:

$$\mathsf{reduce}_R(C)(k) = \mathsf{fold}_\oplus(\iota, M \uplus \{v\}) = \mathsf{fold}_\oplus(\mathsf{fold}_\oplus(\iota, M), \{v\}) = a \oplus v$$

Define delta Δ with $\Delta^-(k) = \{v\}$ and $\Delta^+(k) = \emptyset$. This is valid since $v \in C(k)$. The updated collection is C' with $C'(k) = M$.

By the incremental correctness property:

$$\mathsf{reduce}_R(C')(k) = \mathsf{update}_R(\mathsf{reduce}_R(C)(k), \Delta, k)$$

The left side is:

$$\mathsf{reduce}_R(C')(k) = \mathsf{fold}_\oplus(\iota, M) = a$$

The right side is:

$$\mathsf{update}_R(a \oplus v, \Delta, k) = \mathsf{fold}_\oplus(\mathsf{fold}_\ominus(a \oplus v, \{v\}), \emptyset) = (a \oplus v) \ominus v$$

Therefore $(a \oplus v) \ominus v = a$.

This establishes the inverse property for all a in the image of $\mathsf{fold}_{\oplus}(\iota, -)$. Since every accumulator value that arises during execution of reduce is of this form, this suffices for correctness. □

Remark 3. *The proof of (2 ⇒ 1) establishes the inverse property for* reachable *accumulator values—those expressible as* $\mathsf{fold}_{\oplus}(\iota, M)$ *for some finite multiset* M*. If the accumulator type* A *contains unreachable values, the inverse property need not hold for them. In practice, for reducers like sum over integers or product over rationals, all values are reachable, so the distinction is immaterial.*

6 Examples

6.1 Sum Reducer

$$R_{\mathsf{sum}} = (0, \lambda(a, v).\, a + v, \lambda(a, v).\, a - v)$$

This reducer is well-formed: addition is commutative, and subtraction is the inverse of addition.

Worked example. Suppose for key k we have $C(k) = \{3, 5, 7\}$, so the current view is $a_k = 0 + 3 + 5 + 7 = 15$. Now suppose value 5 is removed and value 2 is added, giving delta $\Delta^-(k) = \{5\}$ and $\Delta^+(k) = \{2\}$. The incremental update computes:

$$a'_k = (15 - 5) + 2 = 12$$

This matches a full recompute: $0 + 3 + 7 + 2 = 12$.

6.2 Count Reducer

$$R_{\mathsf{count}} = (0, \lambda(a, v).\, a + 1, \lambda(a, v).\, a - 1)$$

This reducer counts the number of values, ignoring their content. It is well-formed since $(a + 1) - 1 = a$.

Worked example. For $C(k) = \{x, y, z\}$, we have $a_k = 3$. If y is removed ($\Delta^- = \{y\}$) and w is added ($\Delta^+ = \{w\}$), then:

$$a'_k = (3 - 1) + 1 = 3$$

6.3 Min Reducer (Partial)

$$R_{\mathsf{min}} = (+\infty, \lambda(a, v).\ \min(a, v), \bot)$$

The min reducer does not have a well-defined remove operation $\ominus$ in general. Consider $C(k) = \{3, 5\}$ with $a_k = 3$. If we remove 5, we need $a'_k = 3$, which is correct. But if we remove 3, we need $a'_k = 5$—yet from $a_k = 3$ alone, we cannot recover that 5 was the second-smallest value.

The Skip runtime handles such *partial reducers* by either:

- Recomputing from scratch when values are removed, or
- Maintaining additional state (e.g., a sorted structure of all values).

7 Complexity

Theorem 3 (Time Complexity)**.** *For a well-formed reducer with $O(1)$ add and remove operations:*

- *Adding a value: $O(1)$ to update the accumulator*
- *Removing a value: $O(1)$ to update the accumulator*

This is in contrast to a naïve re-fold which would require $O(n)$ time where n is the size of the multiset.

8 Related Work

The problem of efficiently maintaining aggregations over changing data has been studied extensively. We position Skip's design relative to streaming systems, reactive programming, and incremental computation, and describe our formal contribution.

Skip's design: combinators vs. callbacks. Skip's reducer $R = (\iota, \oplus, \ominus)$ is a *first-class combinator*—a reusable, composable value that can be applied to any reactive collection via reduce_R. This differs from systems that allow users to supply add/remove *callbacks* tied to specific engine internals. Callbacks in other systems are bound to particular contexts (keyed tables, windows, SQL queries); Skip's combinator is context-agnostic and portable across different reactive pipelines.

Streaming systems with add/remove callbacks. Several streaming systems allow users to supply both add and remove logic, but not as a first-class combinator:

Apache Flink (Table API) allows user-defined aggregate functions with `accumulate` and `retract` methods. These are lifecycle methods on a stateful object that Flink's query planner plugs into its retraction-message algebra—not a standalone combinator.

Apache Kafka Streams provides `Aggregator` (add) and `Subtractor` (remove) interfaces for KTable aggregations. These are callbacks passed to the `aggregate()` method, tightly bound to KTable's update mechanism.

Esper CEP supports `enter`/`leave` callbacks for windowed aggregation, but only within sliding-window contexts.

In all cases, the add/remove logic is configuration for a specific operator, not a reusable combinator. Moreover, none of these systems provide formal correctness conditions—they document informally what the remove function should do, but correctness is the user's responsibility.

Systems without user-defined remove. *Apache Beam* and *Spark Streaming* do not expose user-defined inverse operations; they handle retractions internally via recomputation or diffing. *Functional reactive programming* libraries (Fran, Yampa, Reactive Banana [8]) provide append-only fold combinators (`foldp`, `accumE`) with no built-in remove. *Incremental computation* frameworks (Adapton, Jane Street's Incremental) support removal internally via dependency tracking, but do not expose a user-facing inverse operation.

Differential Dataflow and DBSP. Differential dataflow [2] and DBSP [3] provide rigorous foundations for incremental computation using Z-sets (multisets with integer multiplicities) and abelian groups. These frameworks *internally* ensure that all built-in aggregations have inverses, but the user typically works at a higher level (SQL, dataflow graphs) without explicitly defining $\oplus$ and $\ominus$. Skip's design surfaces this algebraic structure as a user-facing combinator; this paper provides the formal analysis.

Database view maintenance. The database literature classifies aggregates as *distributive*, *algebraic*, or *holistic* [6]. Skip's well-formed reducers correspond to *invertible distributive aggregates*. Tangwongsan et al. [7] note that "prior work often relies on aggregation functions to be invertible" for efficient sliding-window maintenance. Yin et al. [9] require inverse functions for incremental graph aggregation. These works focus on algorithmic techniques; our contribution is to formalize correctness for Skip's user-facing abstraction.

Summary: Skip's design and our contribution. Skip provides a first-class reduce combinator $R = (\iota, \oplus, \ominus)$ with user-defined add and remove operations, applicable across arbitrary reactive contexts. Skip's documentation specifies an informal correctness condition for user-defined reducers. Our contribution is to:

1. Formalize this correctness condition as a well-formedness property
2. Prove that correctness holds *if and only if* the property is satisfied (Theorem 2)
3. Connect Skip's design to the theory of invertible distributive aggregates

This gives Skip users a precise specification for writing correct custom reducers, backed by a complete formal characterization.

References

[1] Skip Team. *Skip: A Reactive Programming Framework*. `https://github.com/SkipLabs/skip`, 2024.

[2] Frank McSherry, Derek G. Murray, Rebecca Isaacs, and Michael Isard. Differential dataflow. In *Proceedings of CIDR*, 2013.

[3] Mihai Budiu, Tej Chajed, Frank McSherry, Leonid Ryzhyk, and Val Tannen. DBSP: Automatic incremental view maintenance. In *Proceedings of VLDB*, 16(7):1601–1614, 2023.

[4] Erik Meijer. Your mouse is a database. *Communications of the ACM*, 55(5):66–73, 2012.

[5] Umut A. Acar, Guy E. Blelloch, and Robert Harper. Adaptive functional programming. In *Proceedings of POPL*, pages 247–259, 2002.

[6] Ashish Gupta and Inderpal Singh Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Engineering Bulletin*, 18(2):3–18, 1995.

[7] Kanat Tangwongsan, Martin Hirzel, Scott Schneider, and Kun-Lung Wu. General incremental sliding-window aggregation. *Proceedings of the VLDB Endowment*, 8(7):702–713, 2015.

[8] Conal Elliott and Paul Hudak. Functional reactive animation. In *Proceedings of ICFP*, pages 263–273, 1997.

[9] Shufeng Yin, Huanchen Zhang, Zhengyi Yang, Wentao Han, Wenguang Chen, and Yingxia Shao. GraphBolt: Dependency-driven synchronous processing of streaming graphs. In *Proceedings of ICDE*, 2022.