




Windowed Conflict-Free Replicated Data Types

Wu Tianxing^a , Jonas Spenger^a , and Philipp Haller^a 

^a EECS and Digital Futures, KTH Royal Institute of Technology, Stockholm, Sweden

Abstract **Context** Conflict-free replicated data types (CRDTs) are eventually consistent and highly available replicated data structures. They have found increasing use when data is required to be instantly available for local queries and updates (i.e., high availability), and temporary divergence and eventual convergence of states is tolerated (i.e., eventual consistency). Windowing is a technique used in stateful stream processing systems and has to a large degree contributed to the popularity of these systems. The windowing technique enables slicing infinite streams of updates into finite windows of updates.

Inquiry CRDTs are typically not provided in stream processing systems. They are nondeterministic, which (a) makes it difficult to reason about their processing results and (b) complicates failure recovery. Thus, our work investigates a method for making interactions with CRDTs deterministic.

Approach We introduce *Windowed CRDTs* (WCRDTs), unifying windowed stream computations and CRDTs. WCRDTs can be constructed from arbitrary CRDTs and provide deterministic computations. The WCRDT interface exposed to the stream processor consists of: asynchronous querying and updating of the local CRDT state; incrementing the window; and safely awaiting a window result value, an operation which is potentially blocking.

Knowledge The presented computational model is deterministic. That is, interactions with the windowed CRDTs are guaranteed to provide the same results regardless of nondeterministic processing orders. Additionally, failure recovery is almost trivial to implement; moreover, recovery can be done in parallel or even redundantly to increase reliability. Our evaluation of WCRDTs scale close to linear with the number of processors for reasonable workloads.

Grounding We present a formal model of a stream processing system based on small-step operational semantics. Stream processors evaluate terms from an untyped lambda calculus extended with primitives for interacting with WCRDTs. We prove that the presented model is confluent (informally, deterministic). In addition, we present a prototype implementation of a stream processing system with WCRDTs and failure recovery, and evaluate its performance on a set of microbenchmarks.

Importance WCRDTs are simple to reason about as they provide deterministic results, are highly scalable, and have a simple failure recovery mechanism. Thus, they are well suited for integrating replicated states into stream processing systems, something which typically is not available.

ACM CCS 2012

▪ Theory of computation → Operational semantics; Distributed computing models;

Keywords Conflict-Free Replicated Data Types, Stream Processing Systems, Windowed Computations, Confluence

The Art, Science, and Engineering of Programming

Perspective The Theoretical Science of Programming

Area of Submission Distributed systems programming



© Wu Tianxing, Jonas Spenger, and Philipp Haller
This work is licensed under a “CC BY 4.0” license
Submitted to *The Art, Science, and Engineering of Programming*.

1 Introduction

Conflict-free replicated data types (CRDTs) [27, 30, 31] provide a principled approach to managing highly available and partition-tolerant replicated data in a distributed setting. They do so by relaxing the consistency requirements to strong eventual consistency [33], which allows replicas to temporarily diverge, eventually converge; this is due to a necessary trade-off known as the CAP theorem [7]. The implications of strong eventual consistency, however, are that using CRDTs may result in nondeterministic computations due to the nondeterministic order of synchronizations between replicas. In spite of this, CRDTs have been widely impactful in the software engineering of distributed systems, both in research [3, 31] and practice [21].

The use of CRDTs and other types of replicated data, however, have not been adopted in stream processing systems (SPSs) [15, 32] such as Apache Flink [9], Google Dataflow [2], and Apache Kafka Streams [29]; these are systems that are used to power real-time services at scale [16, 22]. That is, these systems lack abstractions for managing shared replicated data across their processing nodes.

From the point of view of a stream processing system, the nondeterminism of CRDTs is a significant barrier to their adoption. They make it difficult to reason about the results of computations with CRDTs, as well as complicate the failure recovery algorithm. Making interactions with CRDTs deterministic, however, is a non-trivial problem. Recent work has started to address this through related issues such as invariant preservation with CRDTs [18], causal stability / causally-stable operations for CRDTs [4, 5], and on-demand strong consistency for CRDTs [36]. Yet, no prior work has directly addressed the issue of making interactions with CRDTs deterministic.

In an attempt to address this issue, we introduce *windowed CRDTs* (WCRDTs). Windowed CRDTs can be constructed from arbitrary CRDTs and can be used for deterministic computations. The *windowed* in its name refers to windowed computations [2, 35], a well-known technique in SPSs. Windows are used to slice infinite streams of updates into finite windows of updates. In the context of WCRDTs, updates to the CRDT can be grouped into windows, and the results of these updates can be asynchronously synchronized and eventually converge to a deterministic result value per window. From the perspective of a stream processor, a specific window's value can be *awaited*; this is a potentially blocking operation in the case that the window value is not yet available. Besides this, the processor may query and update the local CRDT state, and increment the window; these are non-blocking operations.

In more specific terms, we formally define windowed CRDTs and their operations. Further, we provide a small-step operational semantics for a stream processing system with windowed CRDTs. The stream processors evaluate terms from an untyped lambda calculus extended with primitives for interacting with the windowed CRDTs. We show that the presented semantics is confluent, informally, this means that the computational results are deterministic. In addition to this, we provide a decentralized failure recovery algorithm for the windowed CRDTs. The failure recovery is almost trivial to implement; moreover, recovery can be done in parallel or even redundantly to increase reliability. Unlike failure recovery in Apache Flink [9] and similar systems [15] in which a failure will trigger all processors to recover, our failure recovery algorithm

is decentralized and only triggers the failed processor to recover. Finally, we have also implemented a prototype stream processing system with windowed CRDTs and the described failure recovery algorithm, and evaluated its performance on a set of microbenchmarks. The results show that the windowed CRDTs scale close to linear with the number of processors for reasonable workloads.

Contributions

In summary, this paper makes the following contributions.

- We define and formalize windowed CRDTs together with a recipe for their construction from arbitrary state-based CRDTs (Section 3).
- We provide a small-step operational semantics of windowed CRDTs within a distributed stateful stream processing system (Sections 4, 5). The stream processors evaluate terms in an untyped lambda calculus extended with primitives for windowed CRDTs.
- We prove that the given semantics is confluent and satisfies strong eventual consistency (Section 6).
- We provide an algorithm for failure recovery of windowed CRDTs (Section 7). The algorithm is decentralized and local-first.
- We have implemented a prototype stream processing system with windowed CRDTs and automatic failure recovery, and have evaluated its performance (Section 8). The results show that the windowed CRDTs scale close to linear and that failures do not halt the global execution for reasonable workloads.

We hope that this paper will lead to further work on windowed CRDTs, and ultimately the inclusion of windowed CRDTs in production stream processing systems.

2 Conflict-Free Replicated Data Types (CRDTs)

Conflict-free replicated data types (CRDTs) [27, 30, 31] are particularly useful in a distributed setting in which we require high availability and partition tolerance, and for which we can tolerate a lesser form of consistency known as strong eventual consistency. There are two flavors of CRDTs [31]: state-based and operation-based. They can be shown to be equivalent, and for the purpose of this paper, we will only consider state-based CRDTs.

Informally, a CRDT is a data structure which is replicated across multiple processors. Each processor may perform local updates and queries on the data structure; these updates and queries should be local and non-blocking operations (i.e. high availability, partition tolerance). Asynchronously to the local updates, the data structure replicas should exchange information with each other to synchronize their states. CRDTs provide a principled approach for managing the synchronization, by which the replicas guarantee that they will eventually converge to the same state (strong eventual consistency), i.e. the same query to the replica will return the same result for all replicas. A CRDT can be defined formally like the following [31].

Windowed Conflict-Free Replicated Data Types

Definition 1 (CRDT). A CRDT is a state-based object (S, \leq, s^0, q, u, m) that has the following properties.

1. The set S of payload values forms a semilattice by the partial order \leq .
2. Merging two states s and s' computes their least upper bound, i.e., $s \cdot m(s') = s \sqcup s'$.
3. State is monotonically non-decreasing across updates, i.e., $s \leq s \cdot u$.

The set S and the operations merge (m), query (q), and update (u) are to be defined such that the properties hold. For simplicity, and to avoid confusion in the notation, we may denote $s \cdot m(s')$ as $s \sqcup s'$ in the remainder of the paper.

2.1 CRDT Instances

We will make use of the following two instances of CRDTs in this paper. The first is the add-only set [6], it is a monotonically growing set that satisfies the CRDT properties. The second is the merge map [26], which maps keys to CRDT objects that consequently also satisfy the CRDT properties. To keep things brief, we will only provide the definitions of the add-only set; the definition of the merge map can be found in the appendix.

Definition 2 (Add-Only Set). An add-only set is defined as (S, \leq, s^0, q, u, m) where

$$\begin{aligned} S &\text{ is a set} \\ s \vee s' &= s \cup s' \\ s \cdot u &= s \cup u \\ s^0 &= \emptyset \\ s \leq s' &\iff s \subseteq s' \\ s \cdot q &= s \\ s \cdot m(s') &= s \cup s' \end{aligned}$$

2.2 Guarantees and Limitations

CRDTs guarantee strong eventual consistency. That is, once all replicas have received the same updates, they will have converged to the same value. However, strong eventual consistency cannot guarantee that the final result is independent of the execution order; consequently, this also applies to CRDTs. As this is the core issue that we aim to address in this paper, we will explore this further by example.

Consider the following example program executed by two concurrent processors. The example tries to simulate a situation in which processors share some data, and try to calculate their percentage contribution to the shared data. The program adds value to a shared grow-only counter CRDT, followed by querying it and printing its own contribution in percent.

Listing 1 Program A

```

1 Update_Add 1;
2 Y := Query;
3 Print (100 * 1) / Y;
4 // prints: 25, 50, or 100
5 Update_Add 1;
6 Y := Query;
7 Print (100 * 2) / Y;
8 // prints: 50, 75, or 100

```

Listing 2 Program B

```

1 Update_add 1;
2 Barrier_Sync;
3 Y := Query;
4 Print (100 * 1) / Y;
5 // always prints: 50
6 Update_add 1;
7 Barrier_Sync;
8 Y := Query;
9 Print (100 * 2) / Y;
10 // always prints: 50

```

Listing 3 Program C

```

1 Update_Add 1;
2 Increment_Window;
3 Update_Add 1;
4 Increment_Window;
5 Y := Await Query_Window 0;
6 Print (100 * 1) / Y;
7 // always prints: 50
8 Y := Await Query_Window 1;
9 Print (100 * 2) / Y;
10 // always prints: 50

```

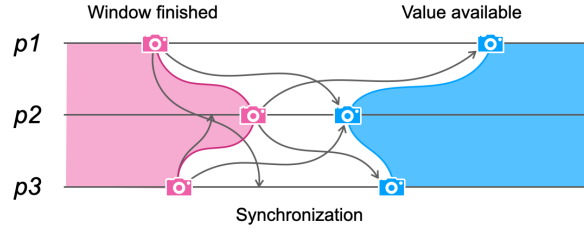
In the case of Program A, we would like to calculate the percentage contribution of each of the two processors, however, due to the nondeterminism, the results may vary greatly. For example, if the querying happens before the underlying synchronization, the query will only return the locally added value, thus the contribution may seem to have been 100 percent. In contrast, if the synchronization happens, we may get varying results, sometimes 25, sometimes 50, depending on whether the other process managed to add and synchronize once or twice before the query. Thus, computations like this may require determinism to be useful, as nondeterministic computations make it difficult to reason about the results. A perhaps naive approach would be to add a barrier that synchronizes the two processors, as is done in Program B. That is, both processors would wait at the barrier until both have reached it before continuing. This will ensure deterministic outcomes, however, it will also unnecessarily block the processors, which will cause the overall execution to slow down. The last approach, Program C, is the one that is explored in this paper. It uses windows to ensure that the processors can safely access the data once the specific window has synchronized. In this way, synchronization and waiting for the result of a window are separated into different operations, thus allowing the processors to continue processing as long as they don't need the result of a window. This captures the main idea, and the following sections will work out the details of this approach.

3 Windowed CRDTs

CRDTs are limited by their eventual consistency giving rise to nondeterministic interactions, as discussed in previous section. A CRDT's data depends on the execution order of the distributed system. Thus, any update of the CRDT based on the current CRDT value may not be consistent among different execution orders. Moreover, strong eventual consistency is not useful in stream processing. As streams are potentially infinite, updates may never stop and effectively prevent consistency to be ever reached. To address these issues, we introduce windowing. Windowing guarantees that, after a window is finished by all processors, it will no longer be modified, and thus eventual consistency can be applied to it.

Figure 1 demonstrates the phases of a window in windowed CRDTs. A window is first finished by every processor (see the red camera symbol), and then starts the asynchronous synchronization via the CRDT mechanism. Once the synchronization

Windowed Conflict-Free Replicated Data Types



■ **Figure 1** The phase of one window in a windowed CRDTs execution.

is finished (the blue camera symbol), the window value would be available to each processor.

3.1 Informal Guarantees

As shown later in Section 6, windowed CRDTs have the following guarantees. Primarily, we show that *windowed CRDTs are confluent* in the context of stream processing for an untyped lambda calculus extended with primitives for windowed CRDTs. This is a nontrivial problem, as discussed in Section 2.2, as programs will usually manage state and branching which may depend on the CRDT state, which in turn may depend on this external program. Further, using confluence, we show that it also has strong eventual consistency, both for the entire execution (i.e., once all replicas have received the same updates, they agree on the same global state), as well as when considering the windows (i.e., once all replicas have received all updates for a window, the replicas will have the same window state). Lastly, we also show that the programming model is *deterministic*, that is, that every window value is independent of processing order.

3.2 Stream Processing System Model

A stream processing system consists of multiple parallel processors, operating on incoming data streams [15, 32]. We formally describe, in this paper, the stream processing system as a set of n stream processors p_1, p_2, \dots, p_n . Each processor has a stream of events delivered to it in order. At any point, the processor may decide that it has finished the previous window and move into a new window. All processors share a global state with windowed CRDT, the user program may interact with windowed CRDT via provided interfaces. Also, the user program is pure or deterministic such that it always behaves the same given the same input.

3.3 Windowed CRDT Object

We define a windowed CRDT object as follows. To note is that a windowed CRDT can be constructed from an arbitrary CRDT S , given some additional metadata, as well as a higher level protocol for managing the metadata in such a way that the guarantees hold.

Definition 3 (Windowed CRDT Object). *Given a CRDT S , a corresponding windowed CRDT object is defined as a tuple (S_{local}, F, Gp, w) where*

- *Local state S_{local} is a record of all local operations on the underlying CRDT not affected by the merge operation.*
- *Finished flag $F \equiv \{p \mapsto w\}$ is a map from processor ID to its latest completed window number.*
- *Global progress $Gp \equiv \{w \mapsto \{p \mapsto (S_{local}, idx, t)\}\}$ is a map from the window number to a collection of each processor's local modifications, message index, and term at the end of that window.*
- *Window number w is the current window being processed (equals to the latest completed window number plus one). We assume that the window number starts from 0.*

A window is considered finished when it is a join of all processors' local states from the moment that they locally finished the given window by incrementing to the next window. S_{local} and w are not affected by other processors and are always accessible for the processor while Gp and F are not.

Definition 4 (CRDT Operation for Windowed CRDT). *For any windowed CRDT object σ , define its CRDT operation as $(\sigma \equiv (S_{local}, F, Gp, w), \leq, \sigma^0, q, u, m)$ where*

$$\begin{aligned} (S_{local}, F, Gp, w) \leq (S'_{local}, F', Gp', w') &\iff \\ F \leq F', Gp \leq Gp' & \\ \sigma^0 = (s^0, \{\}, \{\}, 0) & \end{aligned}$$

$$\begin{aligned} (S_{local}, F, Gp, w) \vee (S'_{local}, F', Gp', w') = \\ (S_{local}, F \sqcup F', Gp \sqcup Gp', w) \end{aligned}$$

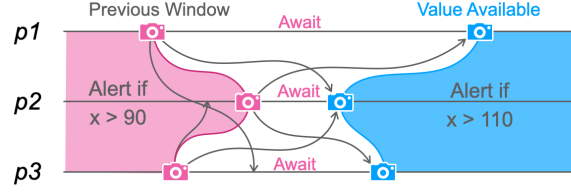
Note that Gp and F are of the presented CRDT types in Section 2. To fully instantiate them, we further require a merge strategy for integers. For this, a simple max-win strategy is used here and we always pick the larger one as the merge result.

3.4 Synchronization and Determinism

To guarantee a stable result regardless of processing order, an update to a CRDT must be independent of its value. However, this is a very strong restriction. In windowed CRDT, such restriction can be relaxed so that the update can only depend on past windows but not current in-progress windows. Thus, the program can make use of the value of the CRDT and its synchronization mechanism.

The asynchronous synchronization in windowed CRDTs happens automatically when a window is finished. The processors will notify each other with a merge message. This process happens concurrently and does not block any message processing. However, when a processor requires some window's value, the processor will have to wait until enough merge messages arrive if it is not available yet so that it can decide the value for that window.

Windowed Conflict-Free Replicated Data Types



■ **Figure 2** Update alerting rules with Windowed CRDT.

3.5 Example

Consider a rule-based alerting system [12] using windowed CRDTs. A website is trying to prevent DDoS attacks. It has 3 servers and wants to detect anomalies in the amount of visitors. The rule is that every window contains 10,000 messages and if the number of visitors is 10% larger than the average value of the last window, it will send an alert to the IT department. Figure 2 shows how a window would be functioning in this situation. We also express the same example in our syntax in Section ??.

The example starts by having an alert set for 90 visitors. Each processor processes messages in the red period of the figure. At the end of the processing of the window demarked by the red photograph image, there are 100 visitors across all processors. This information is asynchronously propagated, and each processor can locally apply the new rule to alert if there are at least 110 visitors once the window has been finalized which occurs at the blue photograph image. The processors continue in the blue period with this new alert rule.

4 Programming Model

We propose a new programming model to capture the interaction between the user program and windowed CRDTs. A program using windowed CRDTs can instantly access the local state S_{local} , and by using an await operation, it gains access to the value of any finalized window. We formalize the model as an extension of untyped lambda calculus, adding several primitives for interaction with windowed CRDTs.

4.1 Syntax

Figure 3 shows the syntactic elements of the programming model. The *unit*, *abstraction* and *application* are standard. The terms *variable*, *unit*, and *abstraction* are values. Terms that are values are also referred to be in a normal form since these terms cannot be further evaluated. The following are specific extensions for windowed CRDTs.

- *state*: Let the processor access its local state (a CRDT) and window number. While allowing the processor use a global state (can only be changed at the end of a window) with *await*, a processor can also utilize a mutable local state within one window, which makes it much more flexible.

$t ::=$		$terms :$
v		$values$
$t t$	$application$	
$localState$	$state$	
$putCRDT\ t; t$	$update$	
$nextWindow; t$	$next$	
$await\ t$	$await$	
$v ::=$		$values :$
x	$variable$	
$()$	$unit$	
$\lambda x. t$	$abstraction$	

■ **Figure 3** Syntax for Windowed CRDTs.

$t ::=$		$terms :$
\dots		
$true$	$true$	
$false$	$false$	
$if\ t\ then\ t\ else\ t$	if	
c	$integer$	
$t + t$	$addition$	
$t - t$	$subtraction$	
$isZero\ t$	$isZero$	
(t, t, \dots, t)	$tuple$	
$t.t$	$index$	
$let\ x = t\ in\ t$	let	

■ **Figure 4** Extensions of Windowed CRDTs' Syntax.

- *update*: Update the processor's local state. Although the system can ensure the update is monotonic by joining it with the old state, a logically correct program should always submit a monotonic update. Otherwise, some updates can be lost.
- *next*: Allow the program to finish the current window and move into the next window.
- *await*: Get the value of a window. It will block until the value is ready. Waiting for a window that one has not reached itself will cause a deadlock.

Some operations like *await* require an integer and others like *putCRDT* and *local* require the ability to encode CRDT as lambda calculus and decode it. These extensions are shown in Figure 4. As these are mostly standard lambda calculus concepts, we leave the details of them to Appendix B.

Windowed Conflict-Free Replicated Data Types

```

1 let (crdt, window) = localState
2 in let (sum, n) = crdt
3 in let average = (await (window - 1)).1 / (await (window - 1)).2
4 in let _ = alert_if (abs (v - average) / average > 0.2)
5 in let _ =
6   -- Go to next window
7   if ???
8   then nextWindow ;
9     putCRDT (sum - sum.value, n - n.value);
10    ()
11 else putCRDT (sum + v, n + 1); ()

```

■ **Listing 4** Term for processor τ .

4.2 Example

In Section 3.5, we consider one possible application of windowed CRDTs, which is a simple rule-based alerting system. Now with the syntax, we can show the exact term that the system would use to process the incoming messages.

Each message contains a value v . And we have a CRDT composed of two PN counters [31] (sum, n) , where *alert* is the threshold of alert, *sum* counts the sum of values in current window and *n* counts the number of messages in the current window.

5 Operational Semantics

Suppose that there are n processors, for a processor k it should have the following fields:

$$\Sigma(k) = (\sigma, i, t)$$

where σ is its windowed CRDT state, i is the index of the current message being processed and t is the term to be evaluated.

To model pending merge messages, we introduce a set of messages in transit,

$$\mathcal{T} \equiv \{(\sigma, k)\}$$

where the windowed CRDT σ is sent to processor k .

There is also a stream of messages that are dispatched to processors, denoted m_i^j where m_i^j is the j th event dispatched to the i th processor.

$$\mathcal{M} = \begin{pmatrix} m_1^1 & m_1^2 & \cdots \\ m_2^1 & m_2^2 & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix}$$

For convenience, we assume that each message is just a user program defined by the syntax. Each message's processing is finished when the current processor term is evaluated to $()$.

We divide the semantics into two classes. One is for global evaluation where the rules operate on a global level. The other one is for local evaluation where local

$$\begin{array}{c}
 \text{EVAL} \frac{k \in [1, n] \quad \Sigma(k).t = t_1 \quad \mathcal{M}, \Sigma, \mathcal{T}, k \mid t_1 \xrightarrow{R} \mathcal{M}, \Sigma', \mathcal{T}', k \mid t_2}{\mathcal{M}, \Sigma, \mathcal{T} \xrightarrow{\text{Eval}(k, R)} \mathcal{M}, [\Sigma(k).t/t_2]\Sigma', \mathcal{T}'} \\
 \\
 \text{NEXTMESSAGE} \frac{k \in [1, n] \quad \Sigma(k) = (\sigma_k, i, ())}{\mathcal{M}, \Sigma, \mathcal{T} \xrightarrow{\text{NextMessage}(k)} \mathcal{M}, [\Sigma(k)/(\sigma_k, i+1, m_k^{i+1})]\Sigma, \mathcal{T}} \\
 \\
 \text{RCV} \frac{(\sigma_i, j) \in \mathcal{T}}{\mathcal{M}, \Sigma, \mathcal{T} \xrightarrow{\text{Rcv}(\sigma_i, j)} \mathcal{M}, [\Sigma(j).\sigma/\Sigma(j).\sigma \sqcup \sigma_i]\Sigma, \mathcal{T} \setminus \{(\sigma_i, j)\}}
 \end{array}$$

■ **Figure 5** Operational Semantics for Windowed CRDT Object (Global).

lambda terms are evaluated. The rule EVAL is used to bring local evaluations to the global context.

Notations used in Semantics We used some notations in the semantics that could cause some confusion. Here is a list of them for clarification.

- $[x \mapsto v]t$ refers to substitution in standard untyped lambda calculus.
- $[t/t_1]\Sigma$ differs to substitution and is used as a shorthand for updating specific values in structured objects. For example, $[\Sigma(k).t/t']\Sigma$ represents replacing $\Sigma(k).t$ with t' such that $([\Sigma(k).t/t']\Sigma)(k).t = t'$, while keeping the rest of the values untouched.
- $\sigma.Gp$ means in variable σ there is a value called Gp , and we want to extract this value.

5.1 Semantics of Global Evaluation

Figure 5 shows evaluation rules that operate on a global level. The rules are in the following form.

$$\mathcal{M}, \Sigma, \mathcal{T} \rightarrow \mathcal{M}, \Sigma', \mathcal{T}'$$

This includes

- **EVAL**: Apply one-step local evaluation R on a certain processor k . It lifts local evaluation semantics into the global evaluation context.
- **NEXTMESSAGE**: Load the next message to a processor with a term $()$. A processor finished processing a message by evaluating it to $()$. The message index for that processor is also incremented.
- **RCV**: Receive a message from \mathcal{T} to the target processor. Message σ_i (a windowed CRDT), intended to deliver to processor j , is removed from the queue \mathcal{T} and joined with processor j 's windowed CRDT replica. Note that by Definition 3, windowed CRDT joins do not affect the local state but update Gp and F which represent other processors' progress.

Windowed Conflict-Free Replicated Data Types

$$\begin{array}{c}
\text{APP1} \frac{\mathcal{M}, \Sigma, \mathcal{T}, k \mid t_1 \rightarrow \mathcal{M}, \Sigma', \mathcal{T}', k \mid t'_1}{\mathcal{M}, \Sigma, \mathcal{T}, k \mid t_1 t_2 \xrightarrow{\text{App1}} \mathcal{M}, \Sigma', \mathcal{T}', k \mid t'_1 t_2} \\
\\
\text{APP2} \frac{\mathcal{M}, \Sigma, \mathcal{T}, k \mid t_2 \rightarrow \mathcal{M}, \Sigma', \mathcal{T}', k \mid t'_2}{\mathcal{M}, \Sigma, \mathcal{T}, k \mid v_1 t_2 \xrightarrow{\text{App2}} \mathcal{M}, \Sigma', \mathcal{T}', k \mid v_1 t'_2} \\
\\
\text{APPABS} \frac{}{\mathcal{M}, \Sigma, \mathcal{T}, k \mid \lambda x. t_1 v_2 \xrightarrow{\text{AppAbs}} \mathcal{M}, \Sigma, \mathcal{T}, k \mid [x \mapsto v_2] t_1} \\
\\
\text{AWAIT} \frac{\begin{array}{c} \text{decode } t = w \\ \sigma_w = \bigsqcup_{i \in [1, n]} \Sigma(k). \sigma. Gp(w)(i). S_{local} \\ \forall i \in [1, n], \Sigma(k). \sigma. F(i) \geq w \end{array}}{\mathcal{M}, \Sigma, \mathcal{T}, k \mid \text{await } t \xrightarrow{\text{Await}(w)} \mathcal{M}, \Sigma, \mathcal{T}, k \mid \text{encode } \sigma_w} \\
\\
\text{NEXTWINDOW} \frac{\begin{array}{c} \Sigma(k) = ((S_{local}, F, Gp, w), idx, t) \\ Gp' = [Gp(w)/Gp(w) \cup \{k \mapsto (S_{local}, idx, t)\}] Gp \\ \Sigma' = [\Sigma(k). \sigma / (S_{local}, [F(k)/w] F, Gp', w + 1)] \Sigma \\ \mathcal{T}_{new} = \{(\Sigma'(k). \sigma, i) \mid i \in [1, n]\} \end{array}}{\mathcal{M}, \Sigma, \mathcal{T}, k \mid \text{nextWindow}; t \xrightarrow{\text{NextWindow}} \mathcal{M}, \Sigma', \mathcal{T} \cup \mathcal{T}_{new}, k \mid t} \\
\\
\text{LOCALSTATE} \frac{t = \text{encode } (\Sigma(k). \sigma. S_{local}, \Sigma(k). \sigma. w)}{\mathcal{M}, \Sigma, \mathcal{T}, k \mid \text{localState} \xrightarrow{\text{LocalState}} \mathcal{M}, \Sigma, \mathcal{T}, k \mid t} \\
\\
\text{UPDATE} \frac{\begin{array}{c} \Sigma(k). \sigma. S_{local} \leq \text{decode } t_1 \\ \Sigma' = [\Sigma(k). \sigma. S_{local} / \Sigma(k). \sigma. S_{local} \sqcup \text{decode } t_1] \Sigma \end{array}}{\mathcal{M}, \Sigma, \mathcal{T}, k \mid \text{putCRDT } t_1 t_2 \xrightarrow{\text{Update}} \mathcal{M}, \Sigma', \mathcal{T}, k \mid t_2}
\end{array}$$

■ **Figure 6** Operational Semantics for Windowed CRDT Object (Local).

5.2 Semantics of Local Evaluation

Figure 6 shows evaluation rules for processor k 's lambda term. The rules are in the following form and are lifted to the global context via EVAL.

$$\mathcal{M}, \Sigma, \mathcal{T}, k \mid t \rightarrow \mathcal{M}, \Sigma', \mathcal{T}', k \mid t'$$

The effect of each rule is shown as follows.

- **APP1, APPABS and APP2:** Standard lambda calculus application and substitution (call by value).
- **AWAIT:** Wait until window w 's value is ready and return the value (a join of all processors' S_{local} for that window). This rule can only be applied when the processor has the value (indicated by every value in $\sigma.F$ reaches at least w).
- **LOCALSTATE:** Return a pair of the processor's local state S_{local} and window number $\sigma.w$ (encoded to lambda calculus terms). It can be instantly evaluated at any point.

- **UPDATE:** Update local state, a CRDT. The new local state is a join of the old value and the provided value to ensure that it is monotonic.
- **NEXTWINDOW:** Complete current window. First, the processor's current local state, message index, and lambda terms are saved to Gp . Second, the record in F for this processor is incremented to mark the completion of the current window and the window number is incremented as well. Last, the updated windowed CRDT σ is broadcast to every processor by inserting the messages to \mathcal{T} .

6 Proof of Guarantees

We limit our analysis to programs that start from an empty initial state, or a state that can be derived from it so that the state is consistent and preserves guarantees windowed CRDTs give. We call a state with such property a valid state.

Definition 5 (Valid Windowed CRDT State). *A windowed CRDT state $\mathcal{M}, \Sigma, \mathcal{T}$ is valid if*

$$\mathcal{M}, \Sigma^0, \emptyset \xrightarrow{*} \mathcal{M}, \Sigma, \mathcal{T}$$

where

$$\Sigma^0 = \{k \mapsto (\sigma^0, 0, ()) \mid k \in [1, n]\}$$

Small-step evaluations can be composed into larger evaluation steps.

Definition 6 (Evaluation Composition). *For a sequence of two evaluation steps*

$$\mathcal{M}, \Sigma, \mathcal{T} \xrightarrow{f} \mathcal{M}, \Sigma', \mathcal{T}' \xrightarrow{g} \mathcal{M}, \Sigma'', \mathcal{T}''$$

Denote the composition of them as

$$\mathcal{M}, \Sigma, \mathcal{T} \xrightarrow{f \circ g} \mathcal{M}, \Sigma'', \mathcal{T}''$$

Evaluation steps are not valid when applied to some states.

Definition 7 (Well-Defined Evaluation Step). *An evaluation step f is well-defined for state $\mathcal{M}, \Sigma, \mathcal{T}$ if there exists a $\mathcal{M}, \Sigma', \mathcal{T}'$ such that*

$$\mathcal{M}, \Sigma, \mathcal{T} \xrightarrow{f} \mathcal{M}, \Sigma', \mathcal{T}'$$

6.1 Confluence

Any valid windowed CRDT initial state $\mathcal{M}, \Sigma, \mathcal{T}$ will be evaluated to the same final state $\mathcal{M}, \Sigma', \mathcal{T}'$ if the evaluation steps are the same but the orders are different. In other words, the evaluation semantics are confluent and commutative.

Definition 8 (Confluence of Windowed CRDT Evaluation). *Windowed CRDT evaluation is confluent if the following property holds. For any valid state $\mathcal{M}, \Sigma, \mathcal{T}$, given $\mathcal{M}, \Sigma, \mathcal{T} \xrightarrow{*} \mathcal{M}, \Sigma_1, \mathcal{T}_1$ and $\mathcal{M}, \Sigma, \mathcal{T} \xrightarrow{*} \mathcal{M}, \Sigma_2, \mathcal{T}_2$ there exists Σ_3 and \mathcal{T}_3 such that*

$$\mathcal{M}, \Sigma_1, \mathcal{T}_1 \xrightarrow{*} \mathcal{M}, \Sigma_3, \mathcal{T}_3$$

$$\mathcal{M}, \Sigma_2, \mathcal{T}_2 \xrightarrow{*} \mathcal{M}, \Sigma_3, \mathcal{T}_3$$

Windowed Conflict-Free Replicated Data Types

Definition 9 (Commutativity of Windowed CRDT Evaluation). *Windowed CRDT evaluation is commutative if for any given $\mathcal{M}, \Sigma, \mathcal{T}$ with two small step evaluation f and g where*

$$\begin{aligned}\mathcal{M}, \Sigma, \mathcal{T} &\xrightarrow{f} \mathcal{M}, \Sigma', \mathcal{T}' \\ \mathcal{M}, \Sigma', \mathcal{T}' &\xrightarrow{g} \mathcal{M}, \Sigma'', \mathcal{T}''\end{aligned}$$

if $f \circ g = g \circ f$ when $f \circ g$ and $g \circ f$ are both well-defined for $\mathcal{M}, \Sigma, \mathcal{T}$.

If every pair of small-step evaluations is commutative, then any two evaluation steps are confluent.

Lemma 1 (Sufficient condition for Confluence). *Windowed CRDT evaluation is confluent if it is commutative.*

Proof. Without loss of generality, let $F = \{f_n \mid n \in [1, i]\}$ and $G = \{g_n, n \in [1, j]\}$ be two evaluations where f_n and g_n are all small step evaluations. That is

$$\begin{aligned}\mathcal{M}, \Sigma, \mathcal{T} &\xrightarrow{f_1 \circ \dots \circ f_i} \mathcal{M}, \Sigma_1, \mathcal{T}_1 \\ \mathcal{M}, \Sigma, \mathcal{T} &\xrightarrow{g_1 \circ \dots \circ g_j} \mathcal{M}, \Sigma_2, \mathcal{T}_2\end{aligned}$$

To prove our lemma, we should find a common evaluation result $\mathcal{M}, \Sigma_3, \mathcal{T}_3$ that can be reached from both $\mathcal{M}, \Sigma_1, \mathcal{T}_1$ and $\mathcal{M}, \Sigma_2, \mathcal{T}_2$, and a series of small step evaluations H that satisfies

$$\mathcal{M}, \Sigma, \mathcal{T} \xrightarrow{H} \mathcal{M}, \Sigma_3, \mathcal{T}_3$$

The simplest way to construct such a H is by combining F and G while removing any duplicated evaluation that exists both in F and G . Consider $H = h_1 \circ h_2 \circ \dots \circ h_k$ such that

$$\forall n_1 \in [1, i] \text{ and } n_2 \in [1, j], f_{n_1} \in H \text{ and } g_{n_2} \in H$$

Thus, we can construct a H as

$$H = \text{deduplicate}(f_1 \circ \dots \circ f_i \circ g_1 \circ \dots \circ g_j)$$

For which *deduplicate* can be defined in such a way. First, if there exists any two *Recv* with the same label, then they are duplicated and only one in F is kept. This operation guarantees that any *Recv* is valid in H . Second, for any k , the evaluation is sequential. F and G may both have a sequence of evaluations (the shorter one will be a prefix of the longer one) on k , and *deduplicate* removes G 's evaluation if it already appears in F 's. As we only remove *Recv* in G , any *Eval(Await)* operation is valid (it must have received at least the same merge messages that it had in F or G). Other local evaluations and *NextMessage* must also be valid since in the second step the longer local sequence is preserved.

Applying commutativity, we have

$$H' = \text{deduplicate}(g_1 \circ \dots \circ g_j \circ f_1 \circ \dots \circ f_i) = H$$

Notice that here F is a prefix of H and G is a prefix of H' . Thus, we have shown that

$$\mathcal{M}, \Sigma_1, \mathcal{T}_1 \xrightarrow{*} \mathcal{M}, \Sigma_3, \mathcal{T}_3$$

$$\mathcal{M}, \Sigma_2, \mathcal{T}_2 \xrightarrow{*} \mathcal{M}, \Sigma_3, \mathcal{T}_3$$

□

Lemma 2. *Windowed CRDT evaluations are commutative.*

Proof. Consider the case that evaluations do not conflict with each other. Some evaluation steps are naturally commutative as their premises rely on disjoint parts of the state and their conclusions modify disjoint parts or the modification is commutative (for instance, two insertions into a set). In such a situation, we can say that they have no conflict and are commutative.

With that, we can conduct a case analysis on every pair of global evaluation rules.

Case 1: $\mathcal{M}, \Sigma, \mathcal{T} \xrightarrow{Eval(i, R_1)} \mathcal{M}, \Sigma', \mathcal{T}' \xrightarrow{Eval(j, R_2)} \mathcal{M}, \Sigma'', \mathcal{T}''$ **where** $i \neq j$

The premises of R_1 and R_2 rely on $\Sigma(i)$ and $\Sigma(j)$ respectively, which is disjoint. R_1 could modify $\Sigma(i)$ and \mathcal{T} , while R_2 could modify $\Sigma(j)$ and \mathcal{T} . $\Sigma(i)$ and $\Sigma(j)$ is disjoint. By the semantics, the only potential modification for local evaluations of \mathcal{T} is inserting new messages (NEXTWINDOW), which is commutative. Thus, in this case, they are commutative for no conflict.

Case 2: $\mathcal{M}, \Sigma, \mathcal{T} \xrightarrow{Eval(i, R_1)} \mathcal{M}, \Sigma', \mathcal{T}' \xrightarrow{Eval(i, R_2)} \mathcal{M}, \Sigma'', \mathcal{T}''$

If $R_1 = R_2$, then it is commutative as it is the same evaluation rule applied twice. Otherwise, as one term only have one corresponding rule to evaluate, $Eval(i, R_2)$ can not be applied first (its premise can not be satisfied). Thus, in this case, it is not well-defined.

Case 3: $\mathcal{M}, \Sigma, \mathcal{T} \xrightarrow{NextMessage(k)} \mathcal{M}, \Sigma', \mathcal{T}' \xrightarrow{R} \mathcal{M}, \Sigma'', \mathcal{T}''$ **and**
 $\mathcal{M}, \Sigma, \mathcal{T} \xrightarrow{R} \mathcal{M}, \Sigma', \mathcal{T}' \xrightarrow{NextMessage(k)} \mathcal{M}, \Sigma'', \mathcal{T}''$

Note that $NextMessage(k)$ only modifies the term and message index in $\Sigma(k)$. We handle this case by a subcase analysis on R .

- Subcase $R = NextMessage(k')$: Whether $k' = k$ or not, inserting messages are commutative.
- Subcase $R = Recv(\sigma, j)$: $Recv$ only changes $\Sigma(k). \sigma$ and \mathcal{T} . Their modifications have on conflict and are thus commutative.
- Subcase $R = Eval(k, R')$: $NextMessage$ requires processor k to have a term $()$, while R will require the processor to have a term that is not $()$. Thus, reorder evaluations in this case will not be well-defined.
- Subcase $R = Eval(k', R')$, $k' \neq k$: In this case, there is no conflict. Their premises rely on $\Sigma(k)$ and $\Sigma(k')$. And their conclusions modify disjoint parts, $\Sigma(k')$, \mathcal{T} and $\Sigma(k)$ respectively.

Case 4: $\mathcal{M}, \Sigma, \mathcal{T} \xrightarrow{Recv(\sigma_1, i)} \mathcal{M}, \Sigma', \mathcal{T}' \xrightarrow{Recv(\sigma_2, j)} \mathcal{M}, \Sigma'', \mathcal{T}''$

Windowed Conflict-Free Replicated Data Types

If $i = j$, since σ is commutative (σ is a semi-join lattice), there is $\sigma_{i_{new}} = \sigma_i \sqcup \sigma_1 \sqcup \sigma_2 = \sigma_i \sqcup \sigma_2 \sqcup \sigma_1$. Thus, $\Sigma'' = [\sigma_{i_{new}}/\sigma]\Sigma$ regardless of evaluation order.

If $i \neq j$, the two evaluations modifies $\Sigma(i)$ and $\Sigma(j)$ and both σ_1 and σ_2 is removed from \mathcal{T} . It is straightforward to show that it is commutative.

Case 5: $\mathcal{M}, \Sigma, \mathcal{T} \xrightarrow{Recv(\sigma, i)} \mathcal{M}, \Sigma', \mathcal{T}' \xrightarrow{Eval(j, R)} \mathcal{M}, \Sigma'', \mathcal{T}''$ and $\mathcal{M}, \Sigma, \mathcal{T} \xrightarrow{Eval(j, R)} \mathcal{M}, \Sigma', \mathcal{T}' \xrightarrow{Recv(\sigma, i)} \mathcal{M}, \Sigma'', \mathcal{T}''$ where $i \neq j$

$Recv(\sigma, i)$ modifies $\Sigma(i)$ and \mathcal{T} , and $Eval(j, R)$ modifies $\Sigma(j)$ and \mathcal{T} (only for $R = NextWindow$). Consider $R \neq NextWindow$, there is no conflict and is thus commutative.

Consider $R = NextWindow$, there could be conflict on \mathcal{T} if $Recv$ removes a message produced by $NextWindow$. However, in that case, it is not possible to evaluate $Recv$ after the interchange of the order and is thus not well-defined. Otherwise, there is no conflict, and is thus commutative.

Case 6: $\mathcal{M}, \Sigma, \mathcal{T} \xrightarrow{Recv(\sigma, i)} \mathcal{M}, \Sigma', \mathcal{T}' \xrightarrow{Eval(i, R)} \mathcal{M}, \Sigma'', \mathcal{T}''$

Notice that lattice join will only affect F and Gp , thus there are only two case worth considering, namely $Recv(\sigma, i) \rightarrow Eval(i, NextWindow)$ and $Recv(\sigma, i) \rightarrow Eval(i, Await(w))$.

In the first case, applying the same argument made in Case 5, we can show that this case is commutative.

For the second case, consider $\Sigma(i).F$. If $F(w) = [1, n]$, $Await$ can be evaluated regardless of $Recv$. Considering that there is no conflict between these two evaluation steps, it is commutative. If $dom(F) \neq [1, n]$, $Await$ can not be evaluated first and is thus not well-defined.

By now, all cases have been proven. □

Theorem 1. *Windowed CRDT evaluation is confluent.*

Proof. This follows directly from Lemma 2 and Lemma 1. □

6.2 Strong Eventual Consistency

Assuming a finite amount of messages, eventually, with no possible further evaluation, all processors have equivalent global progress.

Definition 10 (Strong Eventual Consistency). *Let $\mathcal{M}, \Sigma, \mathcal{T}$ be a valid state. For all evaluation $\mathcal{M}, \Sigma, \mathcal{T} \xrightarrow{*} \mathcal{M}, \Sigma', \mathcal{T}'$, if no evaluation rules can be applied on $\mathcal{M}, \Sigma', \mathcal{T}'$, then there is always*

$$\forall i, j \in [1, n], \Sigma'(i).\sigma.Gp = \Sigma'(j).\sigma.Gp$$

Theorem 2. *Windowed CRDT's evaluation satisfies strong eventual consistency.*

Proof. Consider a special case, every `NEXTWINDOW` is followed by n `Recv`. Suppose this takes x small step evaluation. Notice that for all $i \in [1, n]$, $\Sigma(i).\sigma.Gp =$

$\bigsqcup_{j \in [i, n]} \Sigma(j). \sigma.Gp$. Since $\sigma.Gp$ is a semi-join lattice, there must be $\sigma_1.Gp = \sigma_2.Gp = \dots = \sigma_n.Gp$.

By Theorem 1, all other possible evaluation paths a rewritten of the above case and thus will yield the same conclusion. \square

6.3 Strong Eventual Consistency for Windows

All processors have the same value for a given window after all updates for the window are delivered.

Definition 11 (Strong Eventual Consistency for Windows). *For any valid initial state $\mathcal{M}, \Sigma, \mathcal{T}$ and w , if during evaluation $\mathcal{M}, \Sigma, \mathcal{T} \xrightarrow{*} \mathcal{M}, \Sigma', \mathcal{T}'$ every message inserted by `NEXTWINDOW` for w on each processor has been received via `RECV` then there is*

$$\forall i, j \in [1, n], \Sigma(i). \sigma.Gp = \Sigma(j). \sigma.Gp$$

Theorem 3. *Windowed CRDT's evaluation satisfies strong eventual consistency for windows.*

Proof. By Definition 3, $Gp(w)$ is a map from processor id to each processor's (S_{local}, idx) pair right after the window. Thus, if every merge message for the window is received, every processor must have the same collection (and the collection will have one entry for each processor). \square

6.4 Determinism

All evaluations start from a common valid state, if some steps later, window w has its value to some processors, then the value must be the same for every possible execution path.

Theorem 4 (Determinism for Windowed CRDTs). *For any valid initial state $\mathcal{M}, \Sigma, \mathcal{T}$, let set S be*

$$S = \{\Sigma' \mid \mathcal{M}, \Sigma, \mathcal{T} \xrightarrow{*} \mathcal{M}, \Sigma', \mathcal{T}'\}$$

Then for any $s_1, s_2 \in S$, $k_1, k_2 \in [1, n]$ and $w \in \mathbb{Z}$, there is

$$\forall i \in [1, n], (s_1(k_1).F(i) \geq w \wedge s_2(k_2).F(i) \geq w) \implies s_1(k_1).Gp(w) = s_2(k_2).Gp(w)$$

Proof. We use mathematical induction to prove it. The base case is that there is no `AWAIT` during evaluation.

In this case, processors do not have access to the result of other processors, thus they will always yield the same result for every window. And since Gp is simply a collection of S_{local} , it must be the same.

Then we introduce the induction hypothesis, if the evaluation with $n \geq 0$ `AWAIT` is deterministic, the evaluation with $n + 1$ `AWAIT` is also deterministic.

For any evaluation with $n + 1$ `Await`, there must be a point during evaluation that n `Await` has been evaluated. We know that at this point it is deterministic. Before the $n + 1$ `Await` is evaluated, its window value must have been finished (semantic rules), which is also deterministic. Thus for $n + 1$ `Await`, it must yield a deterministic result and the induction holds. \square

7 Transparent Failure Recovery

In a distributed system, failures are expected to occur regularly and not as singular exceptions [11]. This discussion focuses on crash-silent failures, where processors fail without any warning. Recovering a stateful system from such failures is typically challenging [13, 17], as the internal state may be lost, leading to the corruption of the global state. Failure transparency [34] refers to the system's ability to mask failures from user programs. The system can automatically recover from failures without requiring user intervention.

In windowed CRDTs, windows serve as asynchronous, decentralized snapshots. When the runtime system detects that a processor has failed, it can restore the processor's state from a previously propagated window, ensuring an automatic and transparent recovery.

7.1 Algorithm for Failure Transparent Recovery

To reconstruct the lost state in a failed processor, we can take any other processor and examine its stored window values. We should select the most recent completed window, which contains the state, the message to be processed, and the lambda term it has right at the end of that window. This information is sufficient to reset the processor so it can continue as if it had just finished processing that window. If no completed window is available, we can simply reset the processor to the initial state.

Algorithm 1 Failure Recovery Algorithm

```

1: procedure RECOVER( $i, \mathcal{M}, \Sigma$ )
2:    $j \leftarrow$  random element  $[1, n]$  where  $p_j$  is alive
3:    $w \leftarrow \min \{ \text{codom}(\Sigma(j). \sigma.F) \}$ 
4:    $\Sigma' \leftarrow \Sigma$ 
5:    $\Sigma'(i). \sigma \leftarrow (\Sigma(j).Gp(w)(i).S_{local}, \Sigma(j).F, \Sigma(j).Gp, w)$ 
6:    $\Sigma'(i).i \leftarrow \Sigma(j).Gp(w)(i).idx$ 
7:    $\Sigma'(i).t \leftarrow \Sigma(j).Gp(w)(i).t$ 
8:   return  $\Sigma'$ 
9: end procedure

```

The same procedure to prove confluence for windowed CRDTs (Section 6.1) can be easily applied here.

7.2 Guarantees

The algorithm provides the following guarantees.

Failure is transparent to users. On failures, the user program is reset to some state it experienced in the past so that it continues as if failures have not happened.

Confluence and determinism are preserved. Simliar to reasoning presented in Section 6.1 and Section 6.4. Since we simply reset to processor to a passed state it experienced, if the processor continues, all the window values it gets will stay the same (by induction) and it will follow the same execution steps as it did before the failure. This will result in every single window value stay unchanged. Thus, the system is still confluent and deterministic even with failures and recovery.

8 Implementation and Evaluation

We have implemented a prototype of windowed CRDTs using Scala and the Apache Pekko Actor Model framework [14] framework. We have implemented Windowed CRDTs programming model as presented in Section 4 and the failure handling mechanism described in Section 7.

We evaluated the implemented system as a high-performance computing cluster,¹ with Java 17, Scala 3.4.2, and Pekko 1.0.2. Our evaluation is conducted on AMD EPYC™ 2.25 GHz CPU with 64 cores, along with 256GB RAM.

8.1 Scalability of Windowed CRDTs

Given a failure-free processing scenario, we aim to investigate the scalability of windowed CRDTs.

We divided the experiment into two groups. In the first group, we fixed the number of messages per window and varied the number of windows between two awaits. In the second group, we fixed the number of messages per await while also varying the number of windows between awaits. We repeated these experiments across different numbers of processors. Additionally, we included a baseline where neither windowing nor awaiting occurs, representing the upper bound of CRDT performance. To simulate real-world message processing, we introduced a 100 ms delay for message exchanges between processors, with each message taking 1 ms to process. Each measurement lasts 10 seconds and is repeated 3 times after the warmup phase.

The results are presented in Figure 7. Each group is depicted in a separate plot. The line colors represent the number of messages per window or await, while the line styles indicate the number of windows per await. Based on these figures, we can answer the following questions:

How does performance scale with an increasing number of processors? The performance scales linearly with the number of processors across various setups. In Figure 7a, cases with different numbers of messages per window (∞ , 400, 200) show near-linear growth as the number of processors increases, with only minor differences based on the number of windows per await. Similarly, in Figure 7b, most configurations exhibit

¹ PDC Center for High Performance Computing, KTH Royal Institute of Technology, Stockholm, Sweden.

Windowed Conflict-Free Replicated Data Types

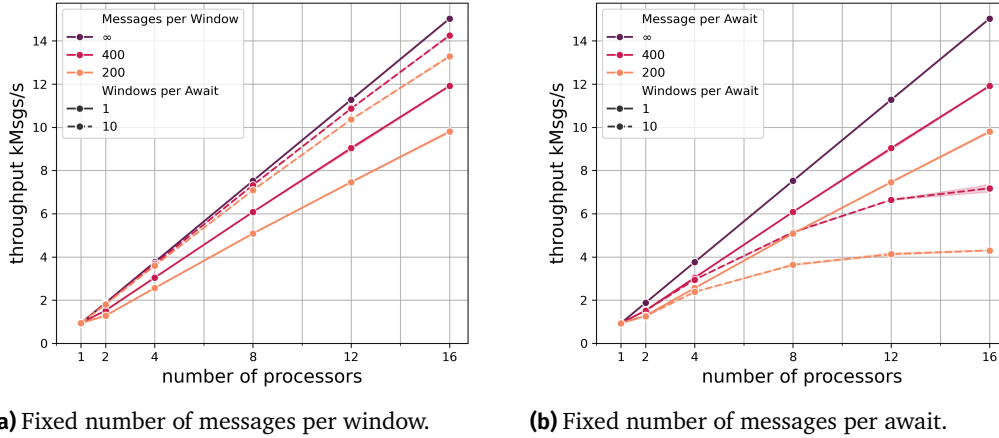


Figure 7 Scalability of windowed CRDTs.

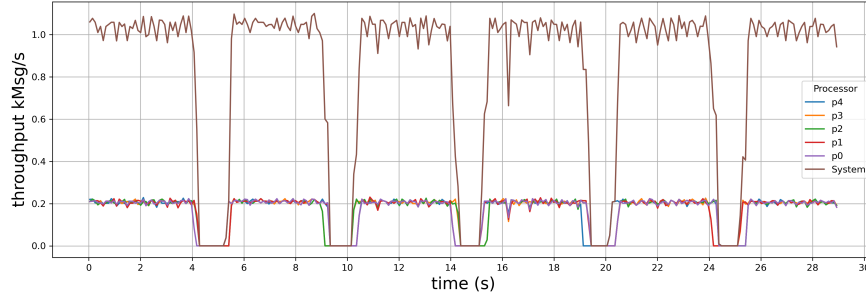
linear scaling, except for extreme conditions (less than 40 messages per window), where the performance plateaus at higher processor counts.

How does the frequency of windowing impact scalability? Increasing the frequency of windowing reduces scalability. Specifically, comparing the solid lines (1 window per await) with the dashed lines (10 windows per await) shows that having more frequent windowing leads to fewer messages per second as the number of processors increases. This effect is especially visible in Figure 7b, where for a fixed number of messages per await, the dashed lines exhibit lower performance than their solid counterparts. Furthermore, this effect is only significant when awaits also occur frequently. In Figure 7a, with few await operations, the lines closely follow the baseline.

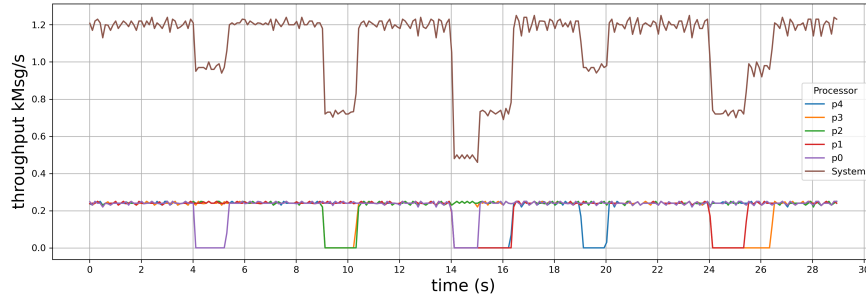
How does the frequency of await operations affect scalability? The frequency of await operations significantly impacts scalability, as shown by the difference in performance between the lines representing different "Messages per Await" values. In both Figure 7a and Figure 7b, the purple lines (baseline) consistently show higher scalability compared to the red (400 messages per await) and orange lines (200 messages per await), which represent more frequent await operations. And within Figure 7a, the dashed lines (10 windows per await or 10 times less await operations) is much closer to baseline than solid lines (1 window per await).

Explanation The reason behind these results is that frequent await will force processors to stop processing and wait for messages to propagate until everyone is lined up with each other. While windowing is an asynchronous operation, await is a blocking call and thus is more impactful on performance.

Conclusion Both windows and awaits can negatively impact throughput, with awaits having a more pronounced effect. Windowed CRDTs scale effectively with the number of processors, provided that frequent windows and awaits do not occur simultaneously.



(a) Throughput over time with await.



(b) Throughput over time without await.

■ **Figure 8** Message Processing Speed over time with Failures.

8.2 Failure Recovery

We set up a system using five processors to simulate real-world processing. Each message takes four milliseconds to process, and in the event of a failure, it takes one second for the thread to restart. The system runs for 30 seconds, and every five seconds, we trigger a failure on some of the processors. The processing speed data is sampled every 100 ms. Since an await operation may cause other processors to wait for the failed one, we designed two groups of experiments: one where all processors perform await at the end of every window, and the other one where each processor operates without any awaits.

The performance line is adjusted to reflect absolute progress. In the event of a failure, the failed processor is reset to the most recent window, losing its prior progress. Its performance remains at zero until it catches up to its position before the failure.

The result is shown in Figure 8, and we can answer the following questions:

Does a failure in one processor impact the execution of other processors? Depend on whether there is a data dependency between the processors. In Figure 8a, processors are forced to wait at the end of every window. Thus once some processors fail, everyone has its throughput drop to zero until the failed ones recover. In Figure 8b, only the failed processors have their throughput dropped to zero, indicating the execution of other processors is not interfered with.

Windowed Conflict-Free Replicated Data Types

How is the system performance affected? Overall performance is only impacted by the failed processors and those that depend on the results of the failed ones. The system throughput in both Figure 8a and Figure 8b shows that it can keep the healthy processors running while recovering failed processors.

Conclusion Windowed CRDTs do not block healthy processors' execution unless they need data from failed processors. Thus, it offers optimal performance within our programming model during recovery.

9 Related Work

To the best of our knowledge, we are the first to write about windowed CRDTs. They are a unification of windowed computations in stream processing [2, 35] and CRDTs [27, 30, 31]. There are however works which have touched upon windows and similar topics. For example, Meiklejohn et al. [24] discuss functional-style aggregations on CRDTs which are compared to sliding window aggregations for stream processing. Our work in contrast provides a principled approach to windowed computations for CRDTs. Another work by Adas and Friedman [1] use CRDTs for sliding window sketches. Window sketches differ in that they are approximations of the data, whilst in our work we are looking at exact results. We consider sliding windows as an interesting future research direction for windowed CRDTs.

Windowed CRDTs, in some sense, provide further guarantees on the data. This has been explored in other work. The Global Sequential Protocol (GSP) by Burckhardt et al. [8] provides means to guarantee eventual consistency and sequential ordering. It utilizes a pending buffer to store modifications that have not been acknowledged but are available locally. The difference to our work is that GSP requires a centralized server-client model while windowed CRDTs utilize a decentralized model. In another work by Zhao and Haller [36], they propose providing additional operations which guarantee observable atomic consistency. In contrast, our work does not directly allow for atomic operations, but instead provides reading snapshot consistent data from the finalized windows. Work on the LoRe programming model by Haas et al. [18] provide verifiable guarantees such as invariant preservation for CRDTs. Although there are some similarities due to the focus on additional guarantees, we focus on providing confluence for windowed computations. Other related work considers the causal stability of messages to determine the stability of data for the purpose of metadata compaction among others [4, 5]. Similarly, a final window can be considered stable messages or data in the CRDT. **JS** *TODO: This paper proves a theorem "confluence" and determinism for a similar programming model with CRDTs. It should be included in our analysis. [25].* ◀

Shared, replicated data is typically not available in a distributed stream processing system. However, there are some exceptions. For example, the PN Counter CRDT can be used within the Hazelcast Jet stream processing system [20]. Another example is a decentralized stream processing system with CRDTs called Hydroflow [28] in the context of the Hydro Project [10, 19] being developed in UC Berkeley. Other

work by Martini and Margara [23] attempt to address safe shared state in dataflow systems via Rust’s type system. Compared to our work, their work is mainly concerned with a single node, multi-threaded scenario and rely on the safety of underlying concurrent data structures. Our work focuses on geographically distributed systems and a functional programming style.

10 Conclusions and Future Work

This paper presents windowed CRDTs as a novel approach to managing replicated data in a distributed stream processing system. Compared to regular CRDTs, we prove that interactions with windowed CRDTs are deterministic (formally, we prove that the semantics are confluent). Further, we provide a local-first failure recovery algorithm for the stream processing system. Additionally, we implement and evaluate the proposed windowed CRDTs as well as their automatic failure recovery. The results show that windowed CRDTs have close to linear scalability and that failures only affect the failed processor and not the global execution for reasonable workloads within a simple microbenchmark. For these reasons, we believe that windowed CRDTs are promising and should be considered for inclusion in stream processing systems.

In future work, we plan to improve on implementation aspects of windowed CRDTs, making them practical for real applications. This includes garbage collection of completed windows [5], and more optimal synchronization such as delta CRDTs [3]. In the context of failure recovery, this includes exploring more strategies, for example exploring how failure recovery is triggered, running the failure recovery in parallel, or even running several replicas in parallel redundantly as active standbys for instant recovery. For windowing, we would also like to explore more complex window types such as sliding windows and incremental windows [2].

Acknowledgements This work was partially funded by Digital Futures under a Research Pairs Consolidator grant (PORTALS) and by the Digital Futures Summer Research Internships Programme. We thank the PDC Center for High Performance Computing at KTH Royal Institute of Technology for providing access to the computing resources for our experiments.

References

- [1] Dolev Adas and Roy Friedman. “Sliding Window CRDT Sketches”. In: *40th International Symposium on Reliable Distributed Systems, SRDS 2021, Chicago, IL, USA, September 20-23, 2021*. IEEE, 2021, pages 288–298. DOI: 10.1109/SRDS53918.2021.00036. URL: <https://doi.org/10.1109/SRDS53918.2021.00036>.
- [2] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. “The Dataflow Model: A Practical Approach to

- Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing”. In: *Proc. VLDB Endow.* 8.12 (2015), pages 1792–1803. DOI: 10.14778/2824032.2824076. URL: <http://www.vldb.org/pvldb/vol8/p1792-Akidau.pdf>.
- [3] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. “Delta state replicated data types”. In: *J. Parallel Distributed Comput.* 111 (2018), pages 162–173. DOI: 10.1016/J.JPDC.2017.08.003. URL: <https://doi.org/10.1016/j.jpdc.2017.08.003>.
- [4] Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. “Pure Operation-Based Replicated Data Types”. In: *CoRR abs/1710.04469* (2017). arXiv: 1710.04469. URL: <http://arxiv.org/abs/1710.04469>.
- [5] Jim Bauwens and Elisa Gonzalez Boix. “From causality to stability: understanding and reducing meta-data in CRDTs”. In: *MPLR ’20: 17th International Conference on Managed Programming Languages and Runtimes, Virtual Event, UK, November 4-6, 2020*. Edited by Stefan Marr. ACM, 2020, pages 3–14. DOI: 10.1145/3426182.3426183. URL: <https://doi.org/10.1145/3426182.3426183>.
- [6] Annette Bieniusa, Marek Zawirski, Nuno M. Preguiça, Marc Shapiro, Carlos Baquero, Valter Balegas, and Sérgio Duarte. “An optimized conflict-free replicated set”. In: *CoRR abs/1210.3368* (2012). arXiv: 1210.3368. URL: <http://arxiv.org/abs/1210.3368>.
- [7] Eric A. Brewer. “Towards robust distributed systems (abstract)”. In: *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing, July 16-19, 2000, Portland, Oregon, USA*. Edited by Gil Neiger. ACM, 2000, page 7. DOI: 10.1145/343477.343502. URL: <https://doi.org/10.1145/343477.343502>.
- [8] Sebastian Burckhardt, Daan Leijen, Jonathan Protzenko, and Manuel Fähndrich. “Global Sequence Protocol: A Robust Abstraction for Replicated Shared State”. In: *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*. Edited by John Tang Boyland. Volume 37. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015, pages 568–590. DOI: 10.4230/LIPICS.ECOOP.2015.568. URL: <https://doi.org/10.4230/LIPICS.ECOOP.2015.568>.
- [9] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. “Apache Flink™: Stream and Batch Processing in a Single Engine”. In: *IEEE Data Eng. Bull.* 38.4 (2015), pages 28–38. URL: <http://sites.computer.org/debull/A15dec/p28.pdf>.
- [10] Alvin Cheung, Natacha Crooks, Joseph M Hellerstein, and Mae Milano. “New directions in cloud programming”. In: *arXiv preprint arXiv:2101.01159* (2021).
- [11] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: simplified data processing on large clusters”. In: *Commun. ACM* 51.1 (2008), pages 107–113. DOI: 10.1145/1327452.1327492. URL: <https://doi.org/10.1145/1327452.1327492>.

- [12] Nick G. Duffield, Patrick Haffner, Balachander Krishnamurthy, and Haakon Ringberg. “Rule-Based Anomaly Detection on IP Flows”. In: *INFOCOM 2009. 28th IEEE International Conference on Computer Communications, Joint Conference of the IEEE Computer and Communications Societies, 19-25 April 2009, Rio de Janeiro, Brazil*. IEEE, 2009, pages 424–432. DOI: 10.1109/INFCOM.2009.5061947. URL: <https://doi.org/10.1109/INFCOM.2009.5061947>.
- [13] E. N. Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. “A survey of rollback-recovery protocols in message-passing systems”. In: *ACM Comput. Surv.* 34.3 (2002), pages 375–408. DOI: 10.1145/568522.568525. URL: <https://doi.org/10.1145/568522.568525>.
- [14] The Apache Software Foundation. *Apache Pekko*. Accessed on 2024-07-25. 2024. URL: <https://pekko.apache.org/>.
- [15] Marios Fragkoulis, Paris Carbone, Vasiliki Kalavri, and Asterios Katsifodimos. “A survey on the evolution of stream processing systems”. In: *VLDB J.* 33.2 (2024), pages 507–541. DOI: 10.1007/S00778-023-00819-8. URL: <https://doi.org/10.1007/s00778-023-00819-8>.
- [16] Yupeng Fu and Chinmay Soman. “Real-time Data Infrastructure at Uber”. In: *SIGMOD ’21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. Edited by Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava. ACM, 2021, pages 2503–2516. DOI: 10.1145/3448016.3457552. URL: <https://doi.org/10.1145/3448016.3457552>.
- [17] Felix C. Gärtner. “Fundamentals of Fault-Tolerant Distributed Computing in Asynchronous Environments”. In: *ACM Comput. Surv.* 31.1 (1999), pages 1–26. DOI: 10.1145/311531.311532. URL: <https://doi.org/10.1145/311531.311532>.
- [18] Julian Haas, Ragnar Mogk, Elena Yanakieva, Annette Bieniusa, and Mira Mezini. “LoRe: A Programming Model for Verifiably Safe Local-first Software”. In: *ACM Trans. Program. Lang. Syst.* 46.1 (2024), 2:1–2:26. DOI: 10.1145/3633769. URL: <https://doi.org/10.1145/3633769>.
- [19] Joseph M Hellerstein, Shadaj Laddad, Mae Milano, Conor Power, and Mingwei Samuel. “Initial Steps Toward a Compiler for Distributed Programs”. In: *Proceedings of the 5th workshop on Advanced tools, programming languages, and PLatforms for Implementing and Evaluating algorithms for Distributed systems*. 2023, pages 1–10.
- [20] Hazelcast Inc. *PN Counter*. <https://docs.hazelcast.com/hazelcast/5.5/data-structures/pn-counter>. [Accessed 24-09-2024].
- [21] Lightbend. *Distributed Data*. <https://doc.akka.io/docs/akka/current/typed/distributed-data.html>. [Accessed 24-09-2024].
- [22] Yancan Mao, Zhanghao Chen, Yifan Zhang, Meng Wang, Yong Fang, Guanghui Zhang, Rui Shi, and Richard T. B. Ma. “StreamOps: Cloud-Native Runtime Management for Streaming Services in ByteDance”. In: *Proc. VLDB Endow.* 16.12 (2023), pages 3501–3514. DOI: 10.14778/3611540.3611543. URL: <https://www.vldb.org/pvldb/vol16/p3501-mao.pdf>.

- [23] Luca De Martini and Alessandro Margara. “Safe Shared State in Dataflow Systems”. In: *Proceedings of the 18th ACM International Conference on Distributed and Event-based Systems, DEBS 2024, Villeurbanne, France, June 24-28, 2024*. ACM, 2024, pages 30–41. DOI: 10.1145/3629104.3666029. URL: <https://doi.org/10.1145/3629104.3666029>.
- [24] Christopher Meiklejohn, Seyed H. Haeri, and Peter Van Roy. “Declarative, sliding window aggregations for computations at the edge”. In: *13th IEEE Annual Consumer Communications & Networking Conference, CCNC 2016, Las Vegas, NV, USA, January 9-12, 2016*. IEEE, 2016, pages 32–37. DOI: 10.1109/CCNC.2016.7444727. URL: <https://doi.org/10.1109/CCNC.2016.7444727>.
- [25] Ragnar Mogk, Joscha Drechsler, Guido Salvaneschi, and Mira Mezini. “A fault-tolerant programming model for distributed interactive applications”. In: *Proceedings of the ACM on Programming Languages* 3.OOPSLA (2019), pages 1–29.
- [26] Nuno M. Preguiça. “Conflict-free Replicated Data Types: An Overview”. In: *CoRR abs/1806.10254* (2018). arXiv: 1806.10254. URL: <http://arxiv.org/abs/1806.10254>.
- [27] Nuno M. Preguiça, Joan Manuel Marquès, Marc Shapiro, and Mihai Letia. “A Commutative Replicated Data Type for Cooperative Editing”. In: *29th IEEE International Conference on Distributed Computing Systems (ICDCS 2009), 22-26 June 2009, Montreal, Québec, Canada*. IEEE Computer Society, 2009, pages 395–403. DOI: 10.1109/ICDCS.2009.20. URL: <https://doi.org/10.1109/ICDCS.2009.20>.
- [28] Mingwei Samuel, Joseph M Hellerstein, and Alvin Cheung. “Hydroflow: A Model and Runtime for Distributed Systems Programming”. PhD thesis. MA thesis. EECS Department, University of California, Berkeley (cited on ..., 2021).
- [29] Matthias J. Sax, Guozhang Wang, Matthias Weidlich, and Johann-Christoph Freytag. “Streams and Tables: Two Sides of the Same Coin”. In: *Proceedings of the International Workshop on Real-Time Business Intelligence and Analytics, BIRTE 2018, Rio de Janeiro, Brazil, August 27, 2018*. Edited by Malú Castellanos, Panos K. Chrysanthis, Badrish Chandramouli, and Shimin Chen. ACM, 2018, 1:1–1:10. DOI: 10.1145/3242153.3242155. URL: <https://doi.org/10.1145/3242153.3242155>.
- [30] Marc Shapiro and Nuno M. Preguiça. “Designing a commutative replicated data type”. In: *CoRR abs/0710.1784* (2007). arXiv: 0710.1784. URL: <http://arxiv.org/abs/0710.1784>.
- [31] Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. “Conflict-Free Replicated Data Types”. In: *Stabilization, Safety, and Security of Distributed Systems - 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings*. Edited by Xavier Défago, Franck Petit, and Vincent Villain. Volume 6976. Lecture Notes in Computer Science. Springer, 2011, pages 386–400. DOI: 10.1007/978-3-642-24550-3_29. URL: https://doi.org/10.1007/978-3-642-24550-3_29.
- [32] Robert Stephens. “A Survey of Stream Processing”. In: *Acta Informatica* 34.7 (1997), pages 491–541. DOI: 10.1007/S002360050095. URL: <https://doi.org/10.1007/S002360050095>.

- [33] Douglas B. Terry, Marvin Theimer, Karin Petersen, Alan J. Demers, Mike Spreitzer, and Carl Hauser. “Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System”. In: *Proceedings of the Fifteenth ACM Symposium on Operating System Principles, SOSP 1995, Copper Mountain Resort, Colorado, USA, December 3-6, 1995*. Edited by Michael B. Jones. ACM, 1995, pages 172–183. DOI: 10.1145/224056.224070. URL: <https://doi.org/10.1145/224056.224070>.
- [34] Aleksey Veresov, Jonas Spenger, Paris Carbone, and Philipp Haller. “Failure Transparency in Stateful Dataflow Systems (Technical Report)”. In: *CoRR* abs/2407.06738 (2024). DOI: 10.48550/ARXIV.2407.06738. arXiv: 2407.06738. URL: <https://doi.org/10.48550/arXiv.2407.06738>.
- [35] Juliane Verwiebe, Philipp M. Grulich, Jonas Traub, and Volker Markl. “Survey of window types for aggregation in stream processing systems”. In: *VLDB J.* 32.5 (2023), pages 985–1011. DOI: 10.1007/S00778-022-00778-6. URL: <https://doi.org/10.1007/s00778-022-00778-6>.
- [36] Xin Zhao and Philipp Haller. “Replicated data types that unify eventual consistency and observable atomic consistency”. In: *J. Log. Algebraic Methods Program.* 114 (2020), page 100561. DOI: 10.1016/J.JLAMP.2020.100561. URL: <https://doi.org/10.1016/j.jlamp.2020.100561>.

A Further CRDT Instances

Definition 12 (Merge Map). A merge map is defined as a tuple $(M \equiv \{k \mapsto s\}, \leq, m^0, q, u, m)$ where

s is a CRDT object

$$\begin{aligned} m \vee m' = & \{k \mapsto m(k) \vee m'(k) \mid \forall k \in \text{dom}(m) \cap \text{dom}(m')\} \\ & \cup \{k \mapsto m(k) \mid \forall k \in \text{dom}(m) \setminus \text{dom}(m')\} \\ & \cup \{k \mapsto m'(k) \mid \forall k \in \text{dom}(m') \setminus \text{dom}(m)\} \end{aligned}$$

$$m \cdot u = m \cup u$$

$$m \leq m' \iff$$

$$\text{dom}(m) \subseteq \text{dom}(m') \ \& \ \forall k, m(k) \leq m'(k)$$

$$m^0 = \{\}$$

Here, we define map lookup operation for map $m \equiv \{k \mapsto s\}$ as

$$m(k) = \begin{cases} \emptyset & \text{if } k \notin \text{dom}(m) \\ \{s\} & \text{if } k \in \text{dom}(m) \end{cases}$$

B Extensions and Encoding for the Syntax

Consider the following definition:

Definition 13 (Integer Encoding and Decoding). *Let id be $\lambda x.x$, define positive integer as church numerals*

$$\begin{aligned} encode\ n &= \lambda f.\lambda x.f^{\circ n} x \\ decode\ n &= k \text{ where } n\ id\ () \xrightarrow{k} () \end{aligned}$$

Immediately, we have $+$ and $-$ defined.

$$\begin{aligned} (+) &= \lambda m.\lambda n.\lambda f.\lambda x.m\ f\ (n\ f\ x) \\ pred &= \lambda n.\lambda f.\lambda x.n(\lambda g.\lambda h.h(g\ f))(\lambda u.x)(\lambda u.u) \\ (-) &= \lambda m.\lambda n.(n\ pred)\ m \end{aligned}$$

Note that there might be cases in which $decode$ will be stuck at a normal form. In which case we can simply terminate it as the program is wrong. In the rest of the paper, we will make the same assumption that the user program is correct.

For convenience, we could also introduce Boolean and Conditionals in the form of church encoding.

Definition 14 (Boolean and Conditionals Encoding). *Define the following terms:*

$$\begin{aligned} true &= \lambda a.\lambda b.a \\ false &= \lambda a.\lambda b.b \\ isZero &= \lambda n.n(\lambda x.false)\ true \\ if &= \lambda p.\lambda a.\lambda b.p\ a\ b \end{aligned}$$

We hereby assume that CRDT can be encoded to integers in binary form and ignore the details of the exact mechanism.

Similarly, tuples can also be encoded as

Definition 15 (Tuple Encoding). *Tuples can be encoded as a simplified church-encoded list.*

$$\begin{aligned} encode(a_1, a_2, \dots, a_{n-1}) &= \\ &\lambda x_1.\lambda x_2.\dots.\lambda x_{n-1}. \\ &\text{if } x_1 \text{ then } a_1, \\ &\text{else if } x_2 \text{ then } a_2 \\ &\dots \\ &\text{else if } x_{n-1} \text{ then } a_{n-1} \\ &\text{else } a_n \end{aligned}$$

For tuple projection, we have


$$t_1.t_2 = \text{if } isZero\ t_2 \text{ then } t_1\ true \text{ else } t_1.(t_2 - 1)$$

Furthermore, we can also introduce a let-binding.


Thus, we can extend the syntax shown in Figure 3 by adding the syntax sugar shown in Figure 4.

About the authors


Wu Tianxing is a master's student at KTH Royal Institute of Technology in Stockholm, Sweden. Contact him at tianxing@kth.se.

 <https://orcid.org/0009-0003-3829-9327>

Jonas Spenger is a PhD student at KTH Royal Institute of Technology in Stockholm, Sweden. Contact him at jspenger@kth.se.

 <https://orcid.org/0000-0002-7119-5234>

Philipp Haller is an associate professor at KTH Royal Institute of Technology in Stockholm, Sweden. Contact him at phaller@kth.se.

 <https://orcid.org/0000-0002-2659-5271>