Verified Time-Aware Stream Processing

Rafael Castro Gonçalves Silva
University of Copenhagen

Dmitriy Traytel
University of Copenhagen

Abstract

Stream processing frameworks provide programming abstractions that allow their users to express the desired time-dependent data analysis. The frameworks organize the computation as a directed graph of interconnected operators that perform event-wise transformations. We tackle the correctness question for programs expressed in this way. To this end, we model (possibly stateful) operators and define their composition, model data streams with time-stamps and watermarks, define reusable, modular operators, and prove their correctness in the Isabelle/HOL proof assistant, taking advantage of its advanced coinductive methods infrastructure. We demonstrate the usefulness of our model by verifying stream processing algorithms computing incremental histograms and relational joins.

2012 ACM Subject Classification Security and privacy → Logic and verification; Computing methodologies → Distributed algorithms; Software and its engineering → Data flow languages

Keywords and phrases stream processing, verification, coinduction, Isabelle/HOL

Digital Object Identifier 10.4230/LIPIcs...

Funding This work is supported by a Novo Nordisk Fonden start package grant (NNF20OC0063462).

1 Introduction

Data stream processing frameworks, such as Apache Flink [10], Apache Samza [18], Apache Spark [38], Google Cloud Dataflow [4], and Timely Dataflow [29, 31, 32] are widely used tools for the low-latency parallel processing of large quantities of data arriving at a high velocity and possibly out of order at a software system. For many developers, these frameworks take the role of both the programming language and the operating system as they provide high-level abstractions that transparently hide much of the complexity of parallel programming.

Most stream processing frameworks follow the dataflow paradigm in which *operators* transform streams of time-stamped events and the entire program, called *dataflow*, consists of a graph of operators that together orchestrate the desired computation. Dataflows are programmed at this *logical* level of abstraction, yet they are deployed and executed in parallel on multiple *workers*, which can be separate processes or even separate machines.

Despite their popularity, stream processing frameworks are notorious for correctness issues [17, 21–23, 36, 37]. The predominant countermeasure to errors in the stream processing (and the wider distributed systems) community is testing [5, 17, 37]. For certain kinds of errors, random testing is effective [26]. Others are hard to detect [21] and require specialized approaches [23]. But no matter how elaborated, testing remains fundamentally incomplete.

How to formally reason about and prove the correctness of programs implemented in these frameworks? To answer this question, in this paper we focus on the logical level where the question becomes: "How to reason about operators and their composition?" More specifically, we use the Isabelle/HOL proof assistant (Section 2) to model operators as coinductive input/output state machines, the event streams they manipulate as lazy lists (i.e., potentially infinite sequences), and to define composition of operators by corecursion (Section 3).

Stream processing frameworks manage out-of-order data streams through *logical time-stamps* and *completeness metadata*. Logical time-stamps do not necessarily represent chronological time; rather, they can encompass application-specific semantics, such as grouping of data items based on different versions or origins. The completeness metadata informs operators when a group is complete, i.e., no further data for this group will be arriving in the future, which may prompt the operator to send final results for this group and clean up its state.

We instantiate our core model with event streams that incorporate partially ordered time-stamps and use watermarks [6] as the completeness metadata (Section 4). We define fundamental correctness notions for such event streams, which we call *monotonicity* (watermark give correct predictions about future events) and *productivity* (all data items are eventually followed by corresponding watermarks). We then identify and prove the correctness of basic, reusable operators for time-dependent batching of data and incremental computations. To prove correctness of an operator, we develop a blueprint consisting of four properties: (1) operator-specific soundness, which describes the desired properties of an operator's output, (2) operator-specific completeness, which characterizes what must be contained in an operator's output, (3) monotonicity preservation, and (4) productivity preservation. The properties (3) and (4) are crucial for reasoning about composed operators.

To validate our model we conduct two case studies (Section 5). First, we define an incremental histogram computation as the composition of our above reusable operators. We show how the building blocks' correctness properties compose to yield the overall correctness property. In addition, we define an optimized incremental histogram computation directly as a single operator and establish the equivalence between the modular and the optimized variant. Second, we use partially ordered time-stamps, which allow us to model operators with multiple inputs as operators with a single input using the disjoint sum order on time-stamps. We use this insight to define and verify a streaming relational join operator using our building blocks.

Our formalization is not tied to any particular stream processing framework, although many of the design choices were inspired by Timely Dataflow [31]. Our work is best understood as a model for specifying and proving correctness of abstract algorithms in the dataflow paradigm. For example, we represent data streams as lazy lists instead of actual data channels, e.g., the network layer is not part of our model. Nonetheless, our operators are executable, in that we can run small examples before engaging with the formal proofs and test our conjectures. Our formalization is publicly available [1]. In summary, we make the following contributions:

- We develop a framework for formally specifying time-aware stream processing programs, which supports the modular composition of building block operators, provides executable specifications, and is validated in two case studies.
- We distill a four-ingredient proof methodology for operator correctness: soundness, completeness, monotonicity preservation, and productivity preservation.
- Our work constitutes a case study in advanced coinductive methods in a proof assistant including mixing friendly corecursion [8] and monadic recursion [19], coinduction up to friends [8] and other generalizations of (co)inductive proofs.

Related Work Our long term objective is to formalize and verify a simplified variant of the Timely Dataflow [32] stream processing framework in Isabelle. The present work arose from the need to specify what Timely Dataflow programs are expected to do, without delving into framework-specific details. This background explains some of our design choices and justifies why we could not reuse an existing model from the literature. As a distinguishing feature, Timely Dataflow uses partially ordered time-stamps to support cyclic dataflows. We inherit this design choice, and while we have not yet formalized cyclic operator graphs, our framework is prepared to do so. Moreover, we benefit from partially ordered time-stamps when modeling operators with multiple inputs. This design choice also requires us to generalize the standard notion of watermarks [6], which traditional stream processing frameworks with totally ordered time-stamps use [4, 10, 18]. After receiving a traditional watermark *wm*, the operator is aware that no event with time-stamp $t < wm$ will be received in the future. This notion of monotonicity [6] inspired our generalization to partial orders shown in Subsection 4.1.

Next, we discuss related pen-and-paper formalizations of stream processing. Kahn networks [16] are often considered as the origin of dataflow programming but lack a notion of time. Hancock et al. [13] develop *stream processors* as a programmable representation of continuous stream transformations. Continuity in this context means that a finite prefix of the output is determined by a finite prefix of the input. Stream processors are a popular example of mixing recursion and corecursion and have been formalized in Agda [3,11] and Isabelle/HOL [9]. They lack a notion of time and the fact that they include the continuity proof in their very structure complicates the definition of time-aware operators. In contrast, our approach decouples the definition of operators from proofs of their desirable properties. Mamouras [27] organizes abstractions of streams as monoids and operators as deterministic stream transducers. We follow a similar approach, albeit being less general (representing streams as lazy lists) and equating transducers with their semantics using a shallow embedding via a coinductive datatype. Mamouras' framework has a notion of time, but restricts time-stamps to be natural numbers.

In the context of proof assistants, the Isabelle formalization of the FOCUS system [34,35] models time-aware, but in-order possibly infinite streams and operators transforming them. Once again, it uses natural numbers as time-stamps. Hinrichsen et al. [15] model the seminal MapReduce framework [12] in Coq. MapReduce processes finite data batches in parallel and without a notion of time. Lochbihler and Hölzl [24] define in Isabelle recursive functions on lazy lists as fixpoints in chain-complete partial orders [19]. We reuse their library of lazy lists.

2 Codatatypes, Coinduction, and Corecursion

We introduce Isabelle's infrastructure for defining and reasoning about coinductive datatypes, short codatatypes [7]. The following codatatype enat of extended natural numbers has the infinity element ESucc (ESucc (ESucc (. . .))). If **datatype** was used instead of **codatatype** below, the infinity element would not exist and the type would be isomorphic to *nat*.

```
codatatype enat = EZero | ESucc enat
```

To represent finite and infinite data streams in a single type, we work with the codatatype of lazy lists, i.e., finite or infinite sequences of elements:

```
codatatype (lset: 'a) llist = lnull: LNil ([#]) | LCons (lhd: 'a) (ltl: 'a llist) (infixr ## 65)
  for map: lmap where ltl LNil = LNil
```

This command introduces the type $'a$ *llist* of potentially infinite sequences along with:

- constructors LNil :: $'a$ *llist*, written [#], and LCons :: $'a \Rightarrow 'a$ *llist* $\Rightarrow 'a$ *llist*, written ##,
- selectors lhd :: $'a$ *llist* $\Rightarrow 'a$ (where lhd [#] = undefined, i.e., a fixed, unspecified value of type $'a$) and ltl :: $'a$ *llist* $\Rightarrow 'a$ *llist* (where ltl [#] = LNil as specified),
- discriminator lnull :: $'a$ *llist* $\Rightarrow$ *bool* returning true iff its argument is [#],
- the functorial action lmap :: $('a \Rightarrow 'b) \Rightarrow 'a$ *llist* $\Rightarrow 'b$ *llist*, and
- the function lset :: $'a$ *llist* $\Rightarrow 'a$ set extracting the set of elements contained in the lazy list.

The **codatatype** command provides a coinduction principle to reason about equality. Proving that two lazy lists are equal reduces to exhibiting a *bisimulation* R that relates them. A bisimulation is a relation that is stable under discriminators and destructors. Formally, we obtain the following coinduction principle for *llist*:

$$R\ lxs\ lys \Longrightarrow (\bigwedge lxs\ lys\ .\ R\ lxs\ lys \Longrightarrow \mathsf{lnull}\ lxs \longleftrightarrow \mathsf{lnull}\ lys\ \wedge \\ \neg\mathsf{lnull}\ lxs \Longrightarrow \neg\mathsf{lnull}\ lys \Longrightarrow \mathsf{lhd}\ lxs = \mathsf{lhd}\ lys \wedge R\ (\mathsf{ltl}\ lxs)\ (\mathsf{ltl}\ lys)) \Longrightarrow lxs = lys \tag{1}$$

Here, $\bigwedge$ is universal quantification. An induction principle for lset membership is also derived:

$$x \in \mathsf{lset}\ lxs \Longrightarrow (\bigwedge\ x\ lxs.\ P\ x\ (x\ \mathtt{\#\#}\ lxs)) \Longrightarrow \\ (\bigwedge x\ lxs\ y\ .\ y \in \mathsf{lset}\ lxs \Longrightarrow P\ y\ lxs \Longrightarrow P\ y\ (x\ \mathtt{\#\#}\ lxs)) \Longrightarrow P\ x\ lxs \tag{2}$$

All functions in Isabelle/HOL must be total. This is guaranteed for terminating recursive functions on datatypes. In contrast, corecursive functions produce elements of codatatypes are total when they are productive, i.e., they always eventually output a codatatype constructor. Corecursive functions are defined with the **corec** command, where the corecursive calls must be guarded by a codatatype constructor, which ensures productivity. For example:

```
corec lshift :: 'a list ⇒ 'a llist ⇒ 'a llist (infixr @@ 65) where
  xs @@ lys = case xs of [] ⇒ lys | x # xs' ⇒ x ## (xs' @@ lys)
```

where [] and infix # are the ordinary constructors of the (finite) list datatype.

Many total corecursive functions involve other operations than the guarding constructor in the context surrounding the corecursive call. An example is $\mathsf{lconcat}$ characterized by:

$$\mathsf{lconcat}\ lxs = \mathtt{case}\ lxs\ \mathtt{of}\ [\mathtt{\#}] \Rightarrow [\mathtt{\#}]\ |\ xs\ \mathtt{\#\#}\ lxs' \Rightarrow xs\ \mathtt{@@}\ \mathsf{lconcat}\ lxs'$$

The **corec** command permits functions to be used freely in definitions if they have been previously registered as (and proved to be) *friendly* [8]. A function is friendly when it preserves productivity in a rather rigid way: it may consume at most one constructor to produce one constructor. Isabelle can automatically prove that lshift is friendly by supplying the (*friend*) option to the above lshift definition (and adjusting it mildly). With lshift registered as a friend, we can define $\mathsf{lconcat}$, although not using the above equation, which lacks a guarding constructor and hides the complication that all lists in xss may be empty (in which case $\mathsf{lconcat}\ xss = [\mathtt{\#}]$). We refer for details to our formalization, which derives the above equation as a lemma.

The coinduction principle (1) is inconvenient for corecursive functions that use friends in their corecursive call contexts. The **corec** command automatically derives the *coinduction up to congruence* principle, which replaces $R\ (\mathsf{ltl}\ xs)\ (\mathsf{ltl}\ ys)$ by $\mathsf{cong}\ R\ (\mathsf{ltl}\ xs)\ (\mathsf{ltl}\ ys)$ in (1), for the congruence closure cong with respect to currently known friends, and thus allows the bisimulation proof to descend under these friends.

Coinductive predicates are definitions as Knaster–Tarski greatest fixpoints on the predicate lattice [33]. They resemble inductive predicates, but allow the introduction rules to be applied infinitely often. Consider the prefix relation on lazy lists:

```
coinductive lprefix :: 'a llist ⇒ 'a llist ⇒ bool where
    lprefix [#] lys
  | lprefix lxs lys ⟹ lprefix (x ## lxs) (x ## lys)
```

Any finite prefix will eventually reach the base case; the coinductive bit is relevant here to ensure that the definition is reflexive, also for infinite lazy lists. Coinductive predicates are accompanied by a coinduction principle. To prove that a coinductive predicate holds for some arguments, we must exhibit a *witness relation* R that relates them. For $\mathsf{lprefix}$, R is a witness relation if for any R-related lazy lists either the left argument is empty, or both arguments are non-empty, their heads equal, and their tails related by either R or $\mathsf{lprefix}$. Formally:

$$R\ lxs\ lys \Longrightarrow (\bigwedge lxs\ lys\ .\ R\ lxs\ lys \Longrightarrow \mathsf{lnull}\ lxs\ \vee \\ (\neg\ \mathsf{lnull}\ lxs \wedge \neg\ \mathsf{lnull}\ lys \Longrightarrow \mathsf{lhd}\ lxs = \mathsf{lhd}\ lys\ \wedge \\ (R\ (\mathsf{ltl}\ lxs)\ (\mathsf{ltl}\ lys) \vee \mathsf{lprefix}\ (\mathsf{ltl}\ lxs)\ (\mathsf{ltl}\ lys)))) \Longrightarrow \mathsf{lprefix}\ lxs\ lys \tag{3}$$

3 Lazy Lists Processors

We introduce operators as a codatatype, give them a semantics as lazy list transformers, define sequential composition, and prove that its correctness with respect to the given semantics.

3.1 Operators on Lazy Lists

We define the codatatype $('i, 'o)$ *op* of operators with a single constructor parametrized by a user defined function, which we call logic, and a lazy list:

$$\textbf{codatatype}\ ('i,'o)\ op = \mathsf{Logic}\ (\mathsf{apply}\colon ('i \Rightarrow ('i,'o)\ op \times 'o\ list))\ (\mathsf{exit}\colon 'o\ llist)$$

The type variable $'i$ represents the operator's inputs, and $'o$ its outputs. The selector apply applies the operator's logic to an input, yielding a pair consisting of a new (evolved) operator and a list of outputs. The new operator can again be applied to another input, etc. The selector exit handles the situation when there are no more inputs. In this case, the operator may produce final outputs. An operator can be seen as an infinitely deep tree branching over $'i$ (which also may be infinite): apply follows a branch, depending on the provided input, to the next node.

It may seem that our operators are stateless, but corecursive functions may have additional parameters representing the state. Our first example operator has state: it buffers data until a buffer limit n is reached, and then it performs a bulk operation with the function f. At the end of the input stream any remaining data in the buffer is output as well:

```
corec bulk_op where bulk_op f n buf = Logic (λe. if length (e # buf) < n
  then (bulk_op f n (e # buf), []) else (bulk_op f n [], f (e # buf))) (llist_of (f buf))
```

We define next how an operator can be applied to an entire, possibly infinite lazy list. We start with the auxiliary function $\mathsf{produce}_1$, which iterates apply until the operator produces a non-empty output $x\ \#\ xs$, in which case $\mathsf{Some}\ (\mathsf{Inl}\ (op', x, xs, lxs'))$ is returned. The operator may fail to produce a non-empty output, either because the input lazy list ends ($\mathsf{Some}\ (\mathsf{Inr}\ op')$), or nothing in the infinite lazy list causes any output to be produced (None).

```
partial_function (option) produce₁ where
  produce₁ op lxs = (case lxs of [#] ⇒ Some (Inr op)
    | h ## lxs' ⇒ (case apply op h of (op', []) ⇒ produce₁ op' lxs'
                                      | (op', x # xs) ⇒ Some (Inl (op', x, xs, lxs'))))
```

The **partial_function** command [19] defines functions by recursion in the option monad (with non-termination represented by None). The command automatically derives an induction principle for the terminating executions of $\mathsf{produce}_1$, which we rewrite as:

$$\begin{aligned}
&\mathsf{produce}_1\ op\ lxs = \mathsf{Some}\ y \Longrightarrow\\
&(\textstyle\bigwedge op\ h\ lxs'\ op'\ zs.\ \mathsf{apply}\ op\ h = (op', []) \Longrightarrow Q\ op'\ lxs'\ zs \Longrightarrow Q\ op\ (h\ \#\#\ lxs')\ zs) \Longrightarrow\\
&(\textstyle\bigwedge op\ h\ x\ xs\ lxs'\ op'.\ \mathsf{produce}_1\ op\ (h\ \#\#\ lxs) = \mathsf{Some}\ (\mathsf{Inl}\ (op', x, xs, lxs')) \Longrightarrow\\
&\quad \mathsf{apply}\ op\ h = (op', x\ \#\ xs) \Longrightarrow Q\ op\ (h\ \#\#\ lxs')\ (\mathsf{Inl}\ (op', x, xs, lxs'))) \Longrightarrow\\
&(\textstyle\bigwedge op.\ Q\ op\ [\#]\ (\mathsf{Inr}\ op)) \Longrightarrow Q\ op\ lxs\ y
\end{aligned} \tag{4}$$

We define produce to corecursively repeat this processes and concatenate the outputs. The following definition requires @@ to be (registered as) friendly.

```
corec produce where produce op lxs = (case produce₁ op lxs of None ⇒ [#]
    | Some (Inr op') ⇒ exit op'
    | Some (Inl (op', x, xs, lxs')) ⇒ x ## (xs @@ produce op' lxs'))
```

3.2 Sequential Composition

The (sequential) composition of two operators means giving the first operator's output as input to the second operator. As our operators output lists, this requires folding the second operator over the output of the first, for which we use the following finite list variant of $\mathsf{produce}$:

$$\begin{array}{l}\textbf{definition}\ \mathsf{fproduce}\ op\ xs = \mathsf{fold}\ (\lambda e\ (op,\ out).\\ \quad \mathtt{let}\ (op',\ out') = \mathsf{apply}\ op\ e\ \mathtt{in}\ (op',\ out\ \mathtt{@}\ out'))\ xs\ (op,\ [])\end{array}$$

The infix @ is the list append function. The composition operator is:

$$\begin{array}{l}\textbf{corec}\ \mathsf{comp\_op}\ \textbf{where}\ \mathsf{comp\_op}\ op_1\ op_2 = \mathsf{Logic}\ (\lambda ev.\\ \quad \mathtt{let}\ (op_1{}',\ out) = \mathsf{apply}\ op_1\ ev;\ \ (op_2{}',\ out') = \mathsf{fproduce}\ op_2\ out\\ \quad \mathtt{in}\ (\mathsf{comp\_op}\ op_1{}'\ op_2{}',\ out'))\ (\mathsf{produce}\ op_2\ (\mathsf{exit}\ op_1))\end{array}$$

The composed exit value iteratively applies op_2 to op_1's exit. The correctness of $\mathsf{comp\_op}$ states that its production corresponds to the functional composition of its arguments' productions: $\mathsf{produce}\ (\mathsf{comp\_op}\ op_1\ op_2)\ lxs = \mathsf{produce}\ op_2\ (\mathsf{produce}\ op_1\ lxs)$. Unfortunately, this equality does hold unconditionally: if op_1 enters an *unproductive* state ($\mathsf{produce}_1$ returns None), then the equality reduces to $[\#] = \mathsf{exit}\ op_2$. Therefore, we add an assumption stating that either op_1 and all its evolutions are eventually productive or op_2's and all its evolutions' exit is $[\#]$. To this end, we introduce another operator that *skips* the first n outputs of an operator, but does not alter that operator's evolution:

$$\begin{array}{l}\textbf{corec}\ \mathsf{skip\_op}\ \textbf{where}\ \mathsf{skip\_op}\ op\ n = \mathsf{Logic}\ (\lambda ev.\ \mathtt{let}\ (op',\ out) = \mathsf{apply}\ op\ ev\ \mathtt{in}\\ \quad \mathtt{if}\ \mathsf{length}\ out < n\ \mathtt{then}\ (\mathsf{skip\_op}\ op'\ (n - \mathsf{length}\ out),\ [])\\ \quad \mathtt{else}\ (op',\ \mathsf{drop}\ n\ out))\ (\mathsf{ldropn}\ n\ (\mathsf{exit}\ op))\end{array}$$

Using ldropn, which drops the first n elements of a lazy list, the correctness of $\mathsf{skip\_op}$ is:

$$\mathsf{produce}\ (\mathsf{skip\_op}\ op\ n)\ lxs = \mathsf{ldropn}\ n\ (\mathsf{produce}\ op\ lxs) \tag{5}$$

We prove (5) by coinduction up to congruence, instantiating the bisimulation candidate R with

$$\lambda l\ r.\ \exists op\ n\ lxs.\ l = \mathsf{produce}\ (\mathsf{skip\_op}\ op\ n)\ lxs \wedge r = \mathsf{ldropn}\ n\ (\mathsf{produce}\ op\ lxs)$$

(Isabelle's $\mathsf{coinduction}$ proof method automatically constructs this canonical instantiation.) Proving that R is a bisimulation up to congruence yields three subgoals about arbitrary R-related lazy lists l and r: (i) $\mathsf{lnull}\ l \longleftrightarrow \mathsf{lnull}\ r$, (ii) $\mathsf{lhd}\ l = \mathsf{lhd}\ r$, and (iii) $\mathsf{cong}\ R\ (\mathsf{ltl}\ l)\ (\mathsf{ltl}\ r)$, where (ii) and (iii) may additionally assume that l and r do not satisfy lnull. All subgoals are proved with lemmas about $\mathsf{produce}_1$ that are proved by its induction principle (4).

Correctness of $\mathsf{comp\_op}$ is then expressed as

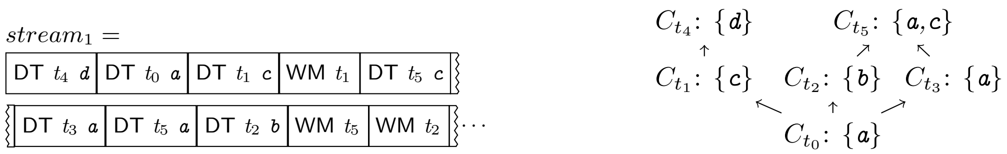
$$\begin{array}{l}(\forall n.\ \mathsf{produce}_1\ (\mathsf{skip\_op}\ op_1\ n)\ lxs \neq \mathsf{None}) \vee (\forall xs.\ \mathsf{exit}\ (\mathsf{fst}\ (\mathsf{fproduce}\ op_2\ xs)) = [\#]) \Longrightarrow\\ \mathsf{produce}\ (\mathsf{comp\_op}\ op_1\ op_2)\ lxs = \mathsf{produce}\ op_2\ (\mathsf{produce}\ op_1\ lxs)\end{array} \tag{6}$$

The proof of (6) also proceeds by coinduction up to congruence. Again, different lemmas about $\mathsf{produce}_1$ must be proved, one relevant example being:

$$\begin{array}{l}\mathsf{produce}_1\ (\mathsf{comp\_op}\ op_1\ op_2)\ lxs = \mathsf{None} \Longrightarrow\\ \mathsf{produce}\ op_1\ lxs = ys\ \mathtt{@@}\ lys \Longrightarrow \exists op_2{}'.\ \mathsf{fproduce}\ op_2\ ys = (op_2{}',\ [])\end{array} \tag{7}$$

We proceed by an induction on ys. Its induction step case is:

$$\begin{array}{l}(\bigwedge op_1\ op_2\ lxs\ lys.\ \mathsf{produce}_1\ (\mathsf{comp\_op}\ op_1\ op_2)\ lxs = \mathsf{None} \Longrightarrow\\ \quad \mathsf{produce}\ op_1\ lxs = ys\ \mathtt{@@}\ lys \Longrightarrow \exists op_2{}'.\ \mathsf{fproduce}\ op_2\ ys = (op_2{}',\ [])) \Longrightarrow\\ \mathsf{produce}_1\ (\mathsf{comp\_op}\ op_1\ op_2)\ lxs = \mathsf{None} \Longrightarrow \mathsf{produce}\ op_1\ lxs = (y\ \#\ ys)\ \mathtt{@@}\ lys \Longrightarrow\\ \exists op_2{}'.\ \mathsf{fproduce}\ op_2\ (y\ \#\ ys)\ [] = (op_2{}',\ [])\end{array}$$



(a) Prefix of $stream_1$

(b) Corresponding set of collections

Figure 1 An example stream and its collections (ordered by their time-stamps)

There is a mismatch between ys in the induction hypothesis and y # ys in the assumption about produce op_1 lxs, so that we cannot use op_1 to instantiate the induction hypothesis. Our solution is to is instantiate op_1 in the induction hypothesis with skip_op op_1 *1*. Using skip_op as a means of generalization is a common pattern in our development. It allows us to formulate statements about arbitrary positions in the output of produce. Namely, when $\mathsf{produce}_1$ (skip_op op n) lxs returns Some (Inl (op, x, xs, lxs)), the value x is op's nth output.

4 Time-Aware Operators

We model time-aware data streams. Events are either time-stamped data items or watermarks:

datatype ($'t$::*order*, $'d$) *event* = DT (tmp: $'t$) (data: $'d$) | WM (tmp: $'t$)

Here, time-stamps $'t$ are assumed to form a partial order via Isabelle's *order* type class. A (time-aware data) stream is a lazy list of type ($'t$, $'d$) *event llist*. A collection C_t is the multiset containing all the data items with the time-stamp t from a time-aware data stream. A watermark WM wm *completes* a time-stamp DT t d if $wm \geq t$. A complete collection is a collection from the lazy list that is completed by some watermark from that lazy list. Figure 1a shows a prefix of an out-of-order stream, named $stream_1$, and Figure 1b is shows the associated set of collections for all time-stamps occurring in the prefix. We arrange the set of collections in a Hasse diagram, partially ordered by the collections' time-stamps. That is in this example, we assume that $t_0 < t_1 < t_4$, $t_0 < t_2 < t_5$, and $t_0 < t_3 < t_5$ are the inequalities that hold for $'t$.

4.1 Monotonicity and Productivity

Watermarks denote that some collection is completed, allowing operators to safely send outputs and clean up their state. If the stream violates the property that after a watermark wm no data with time-stamp $t \leq wm$ will arrive, then operators may no longer work as expected. In other words, the time-stamps on data items must respect a monotonicity property, which intuitively means "never moving backwards" in relation to the already seen watermarks [6].

It is not enough to prohibit time-stamps below the last received watermark due to partially-ordered time-stamps: e.g., given the order from Figure 1b, if WM t_4 was received just after WM t_5 tracking only one of these incomparable time-stamps would necessarily lose information. Instead, we track a set W of received watermarks and ensure that for all wm' in W, every future time-stamp in not less or equal than wm'. Timely Dataflow [32] uses a similar (dual) notion of frontiers, which are antichains of time-stamps (i.e., sets of incomparable time-stamps) that may be encountered in the future. We use plain sets rather than antichains for simplicity.

As streams may be infinite, the monotonicity property is defined coinductively:

coinductive mono **where** mono [#] W
| mono lxs ($\{wm\} \cup W$) $\Longrightarrow$ mono (WM wm ## lxs) W
| ($\forall wm \in W.\ \neg\ wm \geq t$) $\Longrightarrow$ mono lxs W $\Longrightarrow$ mono (DT t d ## lxs) W

The prefix of $stream_1$ shown in Figure 1a satisfies mono $stream_1$ {}.

$$
\begin{array}{l}
\textbf{inductive}\ \mathsf{mono\_cong}\ \textbf{for}\ R\ \textbf{where}\ R\ \mathit{lxs}\ W \Longrightarrow \mathsf{mono\_cong}\ R\ \mathit{lxs}\ W\\
\mid \mathsf{mono\_cong}\ R\ \mathit{lxs}\ (\mathsf{WMs}\ \mathit{xs} \cup W) \Longrightarrow \mathsf{mono}\ (\mathsf{llist\_of}\ \mathit{xs})\ W \Longrightarrow \mathsf{mono\_cong}\ R\ (\mathit{xs}\ @@\ \mathit{lxs})\ W\\
\textbf{inductive}\ \mathsf{prod\_cong}\ \textbf{for}\ R\ \textbf{where}\ R\ \mathit{lxs} \Longrightarrow \mathsf{prod\_cong}\ R\ \mathit{lxs}\\
\mid \mathsf{prod\_cong}\ R\ \mathit{lxs} \Longrightarrow (\forall n < \mathsf{length}\ \mathit{xs}.\ \forall t\ d\ .\ \mathsf{nth}\ \mathit{xs}\ n = \mathsf{DT}\ t\ d \longrightarrow\\
\quad (\exists \mathit{wm} \geq t.\ \mathsf{WM}\ \mathit{wm} \in \mathsf{lset}\ (\mathsf{drop}\ (\mathsf{Suc}\ n)\ \mathit{xs}\ @@\ \mathit{lxs}))) \Longrightarrow \mathsf{prod\_cong}\ R\ (\mathit{xs}\ @@\ \mathit{lxs})
\end{array}
$$

$$
\begin{array}{l}
R\ \mathit{lxs}\ W \Longrightarrow (\bigwedge \mathit{lxs}\ W.\ R\ \mathit{lxs}\ W \Longrightarrow \mathsf{lnull}\ \mathit{lxs} \vee (\exists \mathit{wm}\ \mathit{lxs}'.\ \mathit{lxs} = \mathsf{WM}\ \mathit{wm}\ \#\#\ \mathit{lxs}' \wedge\\
\quad (\mathsf{mono\_cong}\ R\ \mathit{lxs}'\ (\{\mathit{wm}\} \cup W) \vee \mathsf{mono}\ \mathit{lxs}'\ (\{\mathit{wm}\} \cup W))) \vee\\
\ (\exists t\ d\ \mathit{lxs}'.\ \mathit{lxs} = \mathsf{DT}\ t\ d\ \#\#\ \mathit{lxs}' \wedge (\forall \mathit{wm} \in W.\ \neg\, t \leq \mathit{wm}) \wedge\\
\quad (\mathsf{mono\_cong}\ R\ \mathit{lxs}'\ W \vee \mathsf{mono}\ \mathit{lxs}'\ W))) \Longrightarrow \mathsf{mono}\ \mathit{lxs}\ W
\end{array}
\tag{8}
$$

$$
\begin{array}{l}
R\ \mathit{lxs} \Longrightarrow (\bigwedge \mathit{lxs}.\ R\ \mathit{lxs} \Longrightarrow \mathsf{lfinite}\ \mathit{lxs} \vee\\
\ (\exists \mathit{wm}\ \mathit{lxs}'.\ \mathit{lxs} = \mathsf{WM}\ \mathit{wm}\ \#\#\ \mathit{lxs}' \wedge \neg\, \mathsf{lfinite}\ \mathit{lxs}' \wedge (\mathsf{prod\_cong}\ R\ \mathit{lxs}' \vee \mathsf{prod}\ \mathit{lxs}')) \vee\\
\ (\exists t\ d\ \mathit{lxs}'.\ \mathit{lxs} = \mathsf{DT}\ t\ d\ \#\#\ \mathit{lxs}' \wedge \neg\, \mathsf{lfinite}\ \mathit{lxs} \wedge (\exists \mathit{wm} \geq t.\ \mathsf{WM}\ \mathit{wm} \in \mathsf{lset}\ \mathit{lxs}') \wedge\\
\quad (\mathsf{prod\_cong}\ R\ \mathit{lxs}' \vee \mathsf{prod}\ \mathit{lxs}'))) \Longrightarrow \mathsf{prod}\ \mathit{lxs}
\end{array}
\tag{9}
$$

Figure 2 Coinduction up to lshift-congruence principles for mono and prod

Streams must be productive, i.e., always eventually make it possible to produce outputs. We require that for all seen data items DT t d, there eventually is a watermark WM wm with $wm \geq t$. This is crucial for operators' completeness: for every consumed time-stamp there must be an associated output. For finite streams (lfinite), the requirement can be ignored as the stream's end represents the computation's end. We define productivity coinductively:

$$
\begin{array}{l}
\textbf{coinductive}\ \mathsf{prod}\ \textbf{where}\ \mathsf{lfinite}\ \mathit{lxs} \Longrightarrow \mathsf{prod}\ \mathit{lxs}\\
\mid \neg\, \mathsf{lfinite}\ \mathit{lxs} \Longrightarrow (\exists u \geq t.\ \mathsf{WM}\ u \in \mathsf{lset}\ \mathit{lxs}) \Longrightarrow \mathsf{prod}\ \mathit{lxs} \Longrightarrow \mathsf{prod}\ (\mathsf{DT}\ t\ d\ \#\#\ \mathit{lxs})\\
\mid \mathsf{prod}\ \mathit{lxs} \Longrightarrow \mathsf{prod}\ (\mathsf{WM}\ t\ \#\#\ \mathit{lxs})
\end{array}
$$

In Figure 1a, stream_1 may respect this property if it is finite, or if there is a watermark completing t_4 at some later position and the rest of the lazy list respects prod.

The automatically derived coinduction principles for both coinductive predicates are inconvenient for proving some of the properties in Subsection 4.3 due to the occurrence of lshift in produce. The **corec** command [8] derives the coinduction up to congruence principle only for lazy list equality. Therefore, we manually prove coinduction up to congruence for both predicates using their respective regular coinduction principles. To this end, we define inductively in Figure 2 custom congruence closure (under lshift) predicates for both mono and prod. The congruence closure predicates are parameterized by a relation R and allow us to descend under lshift provided that the prefixed list xs preserves the respective properties. In Figure 2, WMs returns the set of all wm values from all watermarks WM wm in a list, nth returns the nth element of a list and drop removes the first n elements of a list. The respective coinduction up to congruence principles (8) and (9) are also shown in Figure 2.

4.2 Building Blocks

We introduce two operators, which we call building blocks, for batching and incremental computations. The batching operator creates *batches*, which are lists of pairs of time-stamps and data, consisting of completed time-stamps that were not part of a previous batch. We introduce an auxiliary function for creating batches given a list of watermarks:

$$
\begin{array}{l}
\textbf{fun}\ \mathsf{batches}\ \textbf{where}\ \mathsf{batches}\ []\ \mathit{tds} = ([],\ \mathit{tds})\\
\mid\ \mathsf{batches}\ (\mathit{wm}\ \#\ \mathit{wms})\ \mathit{tds} = \mathtt{let}\ (\mathit{bs},\ \mathit{tds}') = \mathsf{batches}\ \mathit{wms}\ [(t,\ d) \leftarrow \mathit{tds}.\ \neg\, t \leq \mathit{wm}]\\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathtt{in}\ (\mathsf{DT}\ \mathit{wm}\ [(t,\ d) \leftarrow \mathit{tds}.\ t \leq \mathit{wm}]\ \#\ \mathit{bs},\ \mathit{tds}')
\end{array}
$$

produce (batch_op $[(t_0,a)]$) $stream_1$ =

DT t_1 $[(t_0,a),(t_0,a),(t_1,c)]$	WM t_1	DT t_5 $[(t_2,b),\ (t_3,a),\ (t_5,a),(t_5,c)]$	WM t_5	WM t_2	⋯

Figure 3 A prefix of produce (batch_op $[(t_0,a)]$) $stream_1$

produce (incr_op []) (produce (batch_op $[(t_0,a)]$) $stream_1$) =

DT t_0 $[(t_0,a),(t_0,a),(t_1,c)]$	DT t_1 $[(t_0,a),(t_0,a),(t_1,c)]$	WM t_1

DT t_5 $[(t_0,a),(t_0,a),(t_1,c),(t_2,b),(t_3,a),(t_5,a),(t_5,c)]$	WM t_5	WM t_2	⋯

Figure 4 A prefix of produce (incr_op []) (produce (batch_op $[(t_0,a)]$) $stream_1$)

For every watermark in its first argument, batches computes a batch consisting of pairs from its second argument that have a time-stamp below that watermark. The batch operator is:

```
corec batch_op where
  batch_op tds = Logic (λev. case ev of DT t d ⇒ (batch_op (tds @ [(t, d)]), [])
  | WM wm ⇒ let (out, tds′) = batches [wm] tds
                  in (batch_op tds′, [x ← out. data x ≠ []] @ [WM wm]))
    (let wms = maximal_antichain_list (map fst tds) in llist_of (fst (batches wms tds)))
```

Here, maximal_antichain_list computes a maximal antichain, i.e., a list of distinct, maximal, and incomparable elements from its argument. The operator's buffer *tds* accumulates received DT items and only outputs them (and removes the from the buffer) when a watermark with a greater or equal time-stamp arrives. The end of the input stream is interpreted by the operator's exit as a final watermark which completes all time-stamps in the buffer. Figure 3 shows the output of batch_op $[(t_0,a)]$ after consuming the stream prefix from Figure 1a. The watermark WM t_1 causes the output of a batch containing the collections C_{t_0} and C_{t_1}, whereas WM t_5 outputs all other newly completed collections. If the stream ends at that point, then there will be a final output with DT t_4 $[(t_4,d)]$ caused by the operator's exit.

The incremental computation operator appends batches, creating *accumulated batches*:

```
corec incr_op where
  incr_op abs = Logic (λev. case ev of WM wm ⇒  (incr_op abs, [WM wm])
    | DT wm b ⇒ (incr_op (abs @ b), map (λt. DT t (abs @ b)) (remdups (map fst b)))) [#]
```

Here, remdups removes duplicates in a list. The incr_op operator receives batches and produces batches that are incrementally accumulated. For each time-stamp of the received batch, it outputs a new batch that is concatenated with all previously received batches. The batches are produced without inspecting the time-stamps. Hence, they may include incomparable time-stamps. This design choice simplifies the soundness property. Figure 4 shows the output stream of incr_op [] after consuming the stream prefix from Figure 3.

4.3 Correctness

We identify four *correctness* properties of time-aware operators: soundness, completeness, preservation of monotonicity and preservation of productivity. Soundness means that each operator output meets the operator-specific specification. Completeness means that each input is somehow represented in the operator's output under operator-specific conditions.

For batch_op and incr_op, soundness and completeness statements rely on the auxiliary definitions in Figure 5. There, list_of converts a finite lazy list into a list (and returns undefined for infinite lazy lists), lfilter filters lazy lists, mset transforms a list into a multiset, and ltakeWhile takes the elements from a lazy list white the given predicate holds.

$$
\begin{array}{l}
\textbf{definition}\ \mathsf{map\_filter} :: ('a \Rightarrow 'b\ option) \Rightarrow 'a\ list \Rightarrow 'b\ list\ \textbf{where}\\
\quad \mathsf{map\_filter}\ f\ xs = \mathsf{map}\ (\lambda x\ .\ \mathsf{the}\ (f\ x))\ (\mathsf{filter}\ (\lambda x.\ f\ x \neq \mathsf{None})\ xs)\\
\textbf{definition}\ \mathsf{DTs}\ lxs\ t = \mathsf{map\_filter}\ (\lambda ev.\ \mathtt{case}\ ev\ \mathtt{of}\ \mathsf{DT}\ t\ d \Rightarrow \mathsf{Some}\ d \mid \mathsf{WM}\ wm \Rightarrow \mathsf{None})\\
\quad (\mathsf{list\_of}\ (\mathsf{lfilter}\ (\lambda ev.\ \mathtt{case}\ ev\ \mathtt{of}\ \mathsf{DT}\ t'\ d \Rightarrow t' = t \mid \mathsf{WM}\ wm \Rightarrow \mathsf{False})\ lxs))\\
\textbf{definition}\ \mathsf{lcoll}\ lxs\ t = \mathsf{mset}\ (\mathsf{DTs}\ lxs\ t)\\
\textbf{definition}\ \mathsf{coll}\ xs\ t = \mathsf{mset}\ (\mathsf{map}\ \mathsf{snd}\ (\mathsf{filter}\ (\lambda(t',\ d).\ t' = t)\ xs)\\
\textbf{definition}\ \mathsf{set\_t}\ xs = \mathsf{set}\ (\mathsf{map}\ \mathsf{fst}\ xs)\\
\textbf{definition}\ \mathsf{ts}\ lxs\ t = \{t'.\ \exists d'.\ \ \mathsf{DT}\ t'\ d' \in \mathsf{lset}\ lxs \wedge t' \leq t\}\\
\textbf{definition}\ \mathsf{ws}\ lxs\ wm = \{wm'.\ \mathsf{WM}\ wm' \in \mathsf{lset}\ (\mathsf{ltakeWhile}\ ((\neq)\ (\mathsf{WM}\ wm))\ lxs)\}\\
\textbf{definition}\ \mathsf{incompletes}\ lxs = (\mathtt{let}\ xs = \mathsf{filter}\ (\lambda ev.\ \mathtt{case}\ ev\ \mathtt{of}\\
\qquad \mathsf{DT}\ t\ d \Rightarrow \neg\ (\exists wm \geq t.\ \mathsf{WM}\ wm \in \mathsf{lset}\ lxs) \mid \mathsf{WM}\ \_ \Rightarrow \mathsf{False})\ (\mathsf{list\_of}\ lxs)\\
\quad \mathtt{in}\ \mathsf{maximal\_antichain\_list}\ (\mathsf{map}\ \mathsf{tmp}\ xs))\\
\textbf{definition}\ \mathsf{ws2}\ lxs\ wm = \mathsf{set}\ (\mathsf{takeWhile}\ ((\neq)\ wm)\ (\mathsf{incompletes}\ lxs))\\
\textbf{definition}\ \mathsf{batch\_ts}\ lxs\ wm = \mathtt{if}\ \mathsf{WM}\ wm \in \mathsf{lset}\ lxs\\
\quad \mathtt{then}\ \{t' \in \mathsf{ts}\ lxs\ wm.\ \neg\ (\exists wm' \geq t'.\ wm' \in \mathsf{ws}\ lxs\ wm)\}\\
\quad \mathtt{else}\ \{t' \in \mathsf{ts}\ lxs\ wm.\ \neg\ (\exists wm' \geq t'.\ wm' \in \mathsf{ws2}\ lxs\ wm \vee \mathsf{WM}\ wm' \in \mathsf{lset}\ lxs)\}
\end{array}
$$

Figure 5 Auxiliary definitions for the soundness and completeness of the building blocks

Collections are formalized for both *list* and *llist* types, respectively, by coll and lcoll. As exemplified by Figure 3, the time-stamps of separate batches are disjoint. Under the mono assumption the list_of used in DTs is only applied to finite lazy lists because lfilter is guaranteed to only find finitely many elements. The functions ws and ws2 represent the time-stamps of outputs that possibly happened prior to receiving *wm*. Here, ws2 is the special case for exit, in which there are no actual watermarks completing these time-stamps, so we take the maximal antichain of time-stamps that do not have completing watermarks instead.

4.3.1 Correctness of batch_op.

For batch_op, the intuitive meaning of soundness is: for an output DT *wm b*, the batch *b* is formed by all completed collections with time-stamps $\leq wm$ that were not yet output. We characterize the time-stamps belonging to *b* by batch_ts in Figure 5. Thereby, we distinguish whether the time-stamps are below a watermark from the stream. The full soundness lemma is:

$$
\begin{array}{l}
\mathsf{mono}\ lxs\ W \Longrightarrow \mathsf{DT}\ wm\ b \in \mathsf{lset}\ (\mathsf{produce}\ (\mathsf{batch\_op}\ tds)\ lxs) \Longrightarrow\\
(\forall t \in \mathsf{set\_t}\ b.\ \mathsf{coll}\ b\ t = \mathsf{lcoll}\ lxs\ t + \mathsf{coll}\ tds\ t) \wedge\\
\mathsf{set\_t}\ b = \mathsf{batch\_ts}\ ((\mathsf{map}\ (\lambda\ (t,d).\ \mathsf{DT}\ t\ d)\ tds)\ @@\ lxs)\ wm
\end{array}
\tag{10}
$$

The soundness proof proceeds by induction on lset (2). However, another mismatch with the induction hypothesis forces us to generalize the operator using skip_op. To finish the proof, we use several lemmas about $\mathsf{produce}_1$ that follow by the respective induction principle (4).

Completeness of batch_op states that every input DT *t d* must be present as $(t,\ d)$ in some output batch. From batch_op's soundness (10) we already know that the time-stamps in the output batches form completed collections. Hence we only need to show that the time-stamp is present in some batch. Because of the exit call, we know that every time-stamp eventually will be in some batch, even if there is no watermark completing it. We formally state the completeness of batch_op as follows:

$$
\begin{array}{l}
\mathsf{prod}\ lxs \Longrightarrow \mathsf{DT}\ t\ d \in \mathsf{lset}\ lxs \vee t \in \mathsf{set\_t}\ tds \Longrightarrow\\
\mathsf{lfinite}\ lxs \vee (\forall t' \in \mathsf{set\_t}\ tds.\ \exists wm \geq t'\ .\ \mathsf{WM}\ wm \in \mathsf{lset}\ lxs) \Longrightarrow\\
\exists wm\ b.\ \mathsf{DT}\ wm\ b \in \mathsf{lset}\ (\mathsf{produce}\ (\mathsf{batch\_op}\ tds)\ lxs) \wedge t \in \mathsf{set\_t}\ b
\end{array}
\tag{11}
$$

The third assumption extends the concept of productivity to also include the buffer tds. Completeness follows by induction on lset (2). In the base case, the time-stamp is already in the buffer, or enters the buffer after one call of $\mathsf{produce}_1$. We prove an (omitted) auxiliary lemma that if a time-stamp is in the buffer, it will eventually be part of the output, regardless whether there is a completing watermark in the stream or the stream is finite. The induction step follows by showing that t eventually enters the buffer, which reduces the problem to the auxiliary lemma.

The preservation of mono by batch_op requires an additional assumption about the buffer, which may be output in batch_op's exit, and follows by coinduction up to congruence (8). The preservation prod follows similarly by coinduction up to congruence (9).

$$\begin{array}{l}\mathsf{mono}\ lxs\ W \Longrightarrow (\forall t \in \mathsf{set\_t}\ tds.\ \forall wm \in\ W.\ \neg\ t \le wm) \Longrightarrow \\ \mathsf{mono}\ (\mathsf{produce}\ (\mathsf{batch\_op}\ tds)\ lxs)\ W\end{array} \tag{12}$$

$$\mathsf{prod}\ lxs \Longrightarrow \mathsf{prod}\ (\mathsf{produce}\ (\mathsf{batch\_op}\ tds)\ lxs) \tag{13}$$

4.3.2 Correctness of incr_op.

The soundness for incr_op describes its outputs: an accumulated batch ab which is the concatenation of the buffer with accumulated batches from some prefix of the incoming data stream. The notion of accumulated batches is given by:

```
fun ltake_DT where
  ltake_DT (Suc n) (WM _ ## lxs) = ltake_DT n lxs
| ltake_DT (Suc n) (DT t b ## lxs) = (t, b) # ltake_DT n lxs
| ltake_DT _ _ = []
definition acc_batches n lxs ≡ concat (map snd (ltake_DT n lxs))
```

The soundness of incr_op relates the produced batch DT t ab with the accumulated consumed prefix, and asserts that t originated from the accumulated batches:

$$\begin{array}{l}\mathsf{DT}\ t\ ab \in \mathsf{lset}\ (\mathsf{produce}\ (\mathsf{incr\_op}\ abs)\ lxs) \Longrightarrow \\ \exists n.\ ab = abs\ @\ \mathsf{acc\_batches}\ n\ lxs \wedge t \in \mathsf{set\_t}\ (\mathsf{acc\_batches}\ n\ lxs)\end{array} \tag{14}$$

As for batch_op, soundness requires a generalization with skip_op and proceeds by induction on $\mathsf{produce}_1$ (4) after unfolding produce.

The completeness of incr_op says that every t in every received batch will result in an accumulated batch, and it follows by induction on lset (2):

$$\mathsf{DT}\ wm\ b \in \mathsf{lset}\ lxs \Longrightarrow t \in \mathsf{set\_t}\ b \Longrightarrow \exists ab.\ \mathsf{DT}\ t\ ab \in \mathsf{produce}\ (\mathsf{incr\_op}\ abs)\ lxs \tag{15}$$

The preservation of mono and prod by incr_op also follow, respectively, from their coinduction up to congruence, lemmas (8) and (9), but each require an additional assumption:

$$\begin{array}{l}\mathsf{mono}\ lxs\ W \Longrightarrow (\forall n\ wm\ b.\ n < \mathsf{llength}\ lxs \longrightarrow \mathsf{lnth}\ lxs\ n = \mathsf{DT}\ wm\ b \longrightarrow \\ \quad (\forall t' \in \mathsf{set\_t}\ b\ .\ t' \le wm\ \wedge \\ \qquad (\forall wm' \in\ W \cup \mathsf{vimage}\ \mathsf{WM}\ (\mathsf{lset}\ (\mathsf{ltake}\ n\ lxs)).\ \neg\ wm' \ge t'))) \Longrightarrow \\ \mathsf{mono}\ (\mathsf{produce}\ (\mathsf{incr\_op}\ abs)\ lxs)\ W\end{array} \tag{16}$$

$$\begin{array}{l}\mathsf{prod}\ lxs \Longrightarrow (\forall n\ wm\ b.\ n < \mathsf{llength}\ lxs \longrightarrow \mathsf{lnth}\ lxs\ n = \mathsf{DT}\ wm\ b \longrightarrow \\ \quad (\exists m > n.\ \mathsf{lnth}\ lxs\ m = \mathsf{WM}\ wm) \wedge (\forall t' \in \mathsf{set\_t}\ b.\ t' \le wm) \wedge \neg\ b \ne []) \Longrightarrow \\ \mathsf{prod}\ (\mathsf{produce}\ (\mathsf{incr\_op}\ abs)\ lxs)\end{array} \tag{17}$$

Here, llength is the length function returning an enat, lnth returns the *nth* element of a lazy list, ltake takes the first n::enat elements from a lazy list, and vimage $f\ B = \{x.\ f\ x \in B\}$ is the inverse image of a function. The additional assumptions of the preservation lemmas extend the concepts of mono and prod to the time-stamps inside of the incoming batches.

4.4 Compositional Reasoning

The operators incr_op and batch_op are composable:

definition incr_batch_op *tds abs* = comp_op (batch_op *tds*) (incr_op *abs*)

We prove the four correctness properties for the composed operator by *compositional reasoning*: the proved properties of batch_op prove the assumptions of incr_op. All proofs start by unfolding incr_batch_op and rewriting with comp_op's correctness (6). We note that exit of incr_op always remains [#], which validates the assumption of (6). For preservation of mono, we inherit the additional assumption of (12) which extends monotonicity to the buffer *tds*. Formally:

$$\begin{aligned}&\mathsf{mono}\ \mathit{lxs}\ W \Longrightarrow (\forall t \in \mathsf{set\_t}\ \mathit{tds}.\ \forall\ \mathit{wm} \in\ W.\ \neg\ t \leq \mathit{wm}) \Longrightarrow\\ &\mathsf{mono}\ (\mathsf{produce}\ (\mathsf{incr\_batch\_op}\ \mathit{tds}\ \mathit{abs})\ \mathit{lxs})\ W\end{aligned} \tag{18}$$

Similarly, preservation of productivity is:

$$\mathsf{prod}\ \mathit{lxs} \Longrightarrow \mathsf{prod}\ (\mathsf{produce}\ (\mathsf{incr\_batch\_op}\ \mathit{tds}\ \mathit{abs})\ \mathit{lxs}) \tag{19}$$

Both lemmas are proved by backwards reasoning with the preservation of monotonicity (16) and productivity (17) for incr_op, followed by the preservation of monotonicity (12) and productivity (17) for batch_op, and using batch_op's soundness (10) to discharge the additional assumptions of the preservation lemmas for incr_op.

Each accumulated batch produced by incr_batch_op has some prefix of concatenate DT productions of (batch_op). This fact follows as a corollary of the soundness of batch_op (10) which is then combined with the soundness of incr_op (14) to derive soundness of incr_batch_op:

$$\begin{aligned}&\mathsf{mono}\ \mathit{lxs}\ W \Longrightarrow \mathsf{DT}\ t\ \mathit{ab} \in \mathsf{lset}\ (\mathsf{produce}\ (\mathsf{incr\_batch\_op}\ \mathit{tds}\ \mathit{abs})\ \mathit{lxs}) \Longrightarrow\\ &\exists n.\ \mathit{ab} = \mathit{abs}\ @\ \mathsf{acc\_batches}\ n\ (\mathsf{produce}\ (\mathsf{batch\_op}\ \mathit{tds})\ \mathit{lxs}) \wedge\\ &\quad \{t' \in \mathsf{set\_t}\ (\mathsf{acc\_batches}\ n\ (\mathsf{produce}\ (\mathsf{batch\_op}\ \mathit{tds})\ \mathit{lxs})).\ t' \leq t\} =\\ &\quad \mathsf{ts}\ \mathit{lxs}\ t \cup \{t' \in \mathsf{set\_t}\ \mathit{tds}.\ t' \leq t\} \wedge\\ &\quad (\forall t' \leq t.\ \mathsf{coll}\ (\mathsf{acc\_batches}\ n\ (\mathsf{produce}\ (\mathsf{batch\_op}\ \mathit{tds})\ \mathit{lxs}))\ t' = \mathsf{lcoll}\ \mathit{lxs}\ t' + \mathsf{coll}\ \mathit{tds}\ t')\end{aligned} \tag{20}$$

Completeness of incr_batch_op follows from that of batch_op (11) and incr_op (15):

$$\mathsf{prod}\ \mathit{lxs} \Longrightarrow \mathsf{DT}\ t\ d \in \mathsf{lset}\ \mathit{lxs} \Longrightarrow \exists b.\ \mathsf{DT}\ t\ b \in \mathsf{lset}\ (\mathsf{produce}\ (\mathsf{incr\_batch\_op}\ []\ [])\ \mathit{lxs}) \tag{21}$$

5 Case Study

We introduce two other operators using the building blocks, and showcase how their correctness properties follow compositionally from those of the basic components.

5.1 Histogram

A histogram counts a collection's elements. We compute incremental histograms, which for a time-stamp t count not only elements belonging to the collection C_t, but also elements of collections $C_{t'}$ for all smaller time-stamps $t' < t$. We write H_t for an incremental histogram that counts all elements with time-stamps $\leq t$. We represent (incremental) histograms as multisets.

Partially ordered time-stamps make incremental histograms somewhat counterintuitive. We recall the earlier example of time-stamp collections in Figure 6a. The time-stamp collections C_{t_0}, C_{t_2}, and C_{t_3} result in the incremental histograms H_{t_0}, H_{t_2} and H_{t_3} in Figure 6b. The question is "What precisely should H_{t_5} count and how to compute it?" since it has two immediate predecessor incremental histograms H_{t_2} and H_{t_3} that moreover share H_{t_0} as their predecessor. Here, we use as our specification the variant that counts every included collection only once: $H_{t_5} = C_{t_0} + C_{t_2} + C_{t_3} + C_{t_5}$. A similar example is

$$\begin{tikzcd}[column sep=small]
C_{t_4}\colon \{d\} & & C_{t_5}\colon \{a,c\} & \\
C_{t_1}\colon \{c\} \arrow[u] & C_{t_2}\colon \{b\} \arrow[ur] & & C_{t_3}\colon \{a\} \arrow[ul] \\
& C_{t_0}\colon \{a\} \arrow[ul] \arrow[u] \arrow[urr] & &
\end{tikzcd}$$

(a) Collections in a partial order

$$\begin{tikzcd}[column sep=small]
H_{t_4}\colon \{a,c,d\} & & H_{t_5}\colon \{a,a,a,b,c\} & \\
H_{t_1}\colon \{c,\ a\} \arrow[u] & H_{t_2}\colon \{b,\ a\} \arrow[ur] & & H_{t_3}\colon \{a,\ a\} \arrow[ul] \\
& H_{t_0}\colon \{a\} \arrow[ul] \arrow[u] \arrow[urr] & &
\end{tikzcd}$$

(b) Computed incremental histograms

Figure 6 Computing incremental histograms in partial order

present[1] in the Rust implementation of Differential Dataflow [2, 28, 30], a library for differential computation built on top of Timely Dataflow. There, pairs of numbers are used as the partially ordered time-stamps, whereas we work with an arbitrary partial order. (In our formalization, we additionally verify the more complex specification variant in which $H_{t_5} = H_{t_2} + H_{t_3} + C_{t_5} = C_{t_0} + C_{t_0} + C_{t_2} + C_{t_3} + C_{t_5}$.)

An implementation of our specification using incr_batch_op collects the data with time-stamps $\leq t$ from the accumulated batches at time-stamp t. It needs to compute the histogram for each newly accumulated batch. Without further assumptions on the partial order, any efficient implementation of this incremental histogram specification must keep track of *all* previous incremental histograms, not only maximal ones. For example, in Figure 6a a new collection with time-stamp $t_6 > t_0$ could show up with t_6 being incomparable to all other time-stamps. To compute H_{t_6}, we must thus have access to H_{t_0} (or C_{t_0}).

Our specification sums all collections with time-stamps less or equal than a given t. We use the generic set summation $\mathsf{sum} :: {'a}\ \mathsf{set} \Rightarrow ({'a} \Rightarrow {'b}) \Rightarrow {'b} :: \mathit{comm\_monoid\_add}$ for commutative monoids (here multisets). We also introduce summation for finite accumulated batches.

$$\begin{array}{l}
\textbf{definition}\ \mathsf{S_lcoll}\ \mathit{lxs}\ t = \mathsf{sum}\ (\mathsf{ts}\ \mathit{lxs}\ t)\ (\mathsf{lcoll}\ \mathit{lxs}) \\
\textbf{definition}\ \mathsf{S_coll}\ t\ \mathit{xs} = \mathsf{sum}\ \{t' \in \mathsf{set}\ \mathit{xs}.\ t' \leq t\}\ (\mathsf{coll}\ \mathit{xs})
\end{array}$$

To transform the accumulated batches from incr_batch_op, we use the following time-aware map operator and define the incremental histogram operator by composition:

$$\begin{array}{l}
\textbf{corec}\ \mathsf{map_op}\ \textbf{where}\ \mathsf{map_op}\ f = \mathsf{Logic}\ (\lambda \mathit{ev}.\ \mathsf{case}\ \mathit{ev}\ \mathsf{of} \\
\quad \mathsf{WM}\ \mathit{wm} \Rightarrow (\mathsf{map_op}\ f, [\mathsf{WM}\ \mathit{wm}])\ |\ \mathsf{DT}\ t\ d \Rightarrow (\mathsf{map_op}\ f, [\mathsf{DT}\ t\ (f\ t\ d)])) \\
\textbf{definition}\ \mathsf{ihist_op}\ \mathit{tds}\ \mathit{abs} = \mathsf{comp_op}\ (\mathsf{incr_batch_op}\ \mathit{tds}\ \mathit{abs})\ (\mathsf{map_op}\ \mathsf{S_coll})
\end{array}$$

From the preservation of mono (18) and prod (19) for incr_batch_op and map_op (omitted), both preservation properties are also derived for ihist_op. The soundness and completeness of the incremental histogram, simplified to use empty buffers, are:

$$\mathsf{mono}\ \mathit{lxs}\ W \Longrightarrow \mathsf{DT}\ t\ H \in \mathsf{lset}\ (\mathsf{produce}\ (\mathsf{ihist\_op}\ []\ [])\ \mathit{lxs}) \Longrightarrow H = \mathsf{S\_lcoll}\ \mathit{lxs}\ t \tag{22}$$

$$\mathsf{prod}\ \mathit{lxs} \Longrightarrow \mathsf{DT}\ t\ d \in \mathsf{lset}\ \mathit{lxs} \Longrightarrow \exists H.\ \mathsf{DT}\ t\ H \in \mathsf{lset}\ (\mathsf{produce}\ (\mathsf{ihist\_op}\ []\ [])\ \mathit{lxs}) \tag{23}$$

They follow by incr_batch_op's soundness (20) and completeness (21), respectively.

When time-stamps are natural numbers, an efficient implementation of incremental histograms will compute each H_t by summing the *last stored incremental histogram* H_{t-1} with the newest completed time-stamp collection C_t after seeing WM t. This is efficient because the last histogram has been already computed. In contrast, our generic incremental histogram is inefficient: it has a buffer that is never cleaned, and it recomputes the entire histogram for each accu-

[1] `https://github.com/TimelyDataflow/differential-dataflow/blob/250e9c2ad2e9/examples/multitemporal.rs`

$$
\begin{array}{l}
\textbf{corec } \mathsf{ihist\_op'} \textbf{ where } \mathsf{ihist\_op'}\ H\ \mathit{buf} = \mathsf{Logic}\ (\lambda ev.\ \mathtt{case}\ ev\ \mathtt{of} \\
\quad \mathsf{DT}\ (t{::}\_{::}\mathit{linorder})\ d \Rightarrow (\mathsf{ihist\_op'}\ H\ (\mathit{buf}\ @\ [(t,\ d)]),\ []) \\
\mid \mathsf{WM}\ wm \Rightarrow \mathtt{if}\ \exists(t,\ d) \in \mathsf{set}\ \mathit{buf}\ .\ t \le wm \\
\quad \mathtt{then\ let}\ out = [(t,\ \_) \leftarrow \mathit{buf}.\ t \le wm];\ \mathit{buf}' = [(t,\ \_) \leftarrow \mathit{buf}.\ t > wm]; \\
\quad tss = \mathsf{remdups}\ ((\mathsf{map}\ \mathsf{fst}\ out)); \\
\quad Hs = \mathsf{map}\ (\lambda t.\ \mathsf{DT}\ t\ (H + (\mathsf{mset}\ (\mathsf{map}\ \mathsf{snd}\ [(t'\ \_) \leftarrow out.\ t' \le t])))) \ tss\ \mathtt{in} \\
\qquad (\mathsf{ihist\_op'}\ (H + (\mathsf{mset}\ (\mathsf{map}\ \mathsf{snd}\ out)))\ \mathit{buf}',\ Hs\ @\ [\mathsf{WM}\ wm]) \\
\quad \mathtt{else}\ (\mathsf{ihist\_op'}\ H\ \mathit{buf},\ [])) \\
\ \ (\mathsf{llist\_of}\ (\mathsf{map}\ (\lambda t.\ \mathsf{DT}\ t\ (H + \mathsf{mset}\ (\mathsf{map}\ \mathsf{snd}\ [(t',\ \_) \leftarrow \mathit{buf}.\ t' \le t]))) \\
\quad (\mathsf{remdups}\ (\mathsf{map}\ \mathsf{fst}\ \mathit{buf}))))
\end{array}
$$

Figure 7 Efficient incremental histogram operator

$$\mathsf{produce}\ (\mathsf{ihist\_op'}\ []\ [])\ \boxed{\mathsf{DT}\ \mathit{1}\ a}\boxed{\mathsf{WM}\ \mathit{1}}\boxed{\mathsf{DT}\ \mathit{0}\ a}\boxed{\mathsf{WM}\ \mathit{0}} = \boxed{\mathsf{DT}\ \mathit{1}\ \{a\}}\boxed{\mathsf{WM}\ \mathit{1}}\boxed{\mathsf{DT}\ \mathit{0}\ \{a,\ a\}}\boxed{\mathsf{WM}\ \mathit{0}}$$

$$\mathsf{produce}\ (\mathsf{ihist\_op}\ []\ [])\ \boxed{\mathsf{DT}\ \mathit{1}\ a}\boxed{\mathsf{WM}\ \mathit{1}}\boxed{\mathsf{DT}\ \mathit{0}\ a}\boxed{\mathsf{WM}\ \mathit{0}} = \boxed{\mathsf{DT}\ \mathit{1}\ \{a\}}\boxed{\mathsf{WM}\ \mathit{1}}\boxed{\mathsf{DT}\ \mathit{0}\ \{a\}}\boxed{\mathsf{WM}\ \mathit{0}}$$

Figure 8 Difference between $\mathsf{ihist\_op'}$ and $\mathsf{ihist\_op}$

mulated batch from scratch. We optimize the incremental histogram computation, in Figure 7, for a totally ordered (*linorder* typeclass) time-stamp type. The optimized operator has two buffers: the first one keeps the last histogram (which makes sense for a total order); the second one keeps the tuples not output yet. When new histograms are output, the operator replaces the last stored histogram with the most recent one. The two implementations are equivalent when applied to streams with totally ordered time-stamps respecting mono. Figure 8 demonstrates that the implementations differ when the input is not monotone. The $\mathsf{ihist\_op'}$ operator counts all previously seen events which it stores in its state H, whereas ihist_op is aware of all time-stamps because it uses accumulated batches including all previous time-stamps.

We show that the two operators produce the same results for monotone inputs:

$$
\begin{array}{l}
\mathsf{mono}\ lxs\ W \Longrightarrow (\forall t \in \mathsf{set\_t}\ abs.\ \exists wm \in\ W.\ t \le wm) \Longrightarrow \\
(\forall t \in \mathsf{set\_t}\ tds.\ \forall wm \in\ W.\ t > wm) \Longrightarrow \\
\mathsf{produce}\ (\mathsf{ihist\_op'}\ (\mathsf{mset}\ (\mathsf{map}\ \mathsf{snd}\ abs))\ tds)\ lxs = \mathsf{produce}\ (\mathsf{ihist\_op}\ tds\ abs)\ lxs
\end{array} \tag{24}
$$

Instead of directly proving this equality, we define a coinductive predicate between operators that holds if they produce the same outputs when applied to the same monotone input. The relation, similarly to mono, uses a set W of seen watermarks, and coinductively checks that the time-stamp of the next event is not smaller or equal than any time-stamp in W:

$$
\begin{array}{l}
\textbf{definition } \mathsf{eq\_op\_lifted}\ ev\ W\ op_1\ op_2\ R = \mathsf{exit}\ op_1 = \mathsf{exit}\ op_2 \wedge (\mathtt{case}\ ev\ \mathtt{of} \\
\quad \mathsf{DT}\ t\ d \Rightarrow (\forall wm \in\ W.\ \neg\ t \le wm) \Longrightarrow \mathsf{rel\_prod}\ (R\ \ W)\ (=)\ (\mathsf{apply}\ op_1\ \ ev)\ (\mathsf{apply}\ op_2\ \ ev) \\
\mid \mathsf{WM}\ wm \Rightarrow \mathsf{rel\_prod}\ (R\ (\{wm\} \cup\ W))\ (=)\ (\mathsf{apply}\ op_1\ \ ev)\ (\mathsf{apply}\ op_2\ \ ev)) \\
\textbf{coinductive } \mathsf{eq\_op} \textbf{ where } (\forall ev.\ \mathsf{eq\_op\_lifted}\ ev\ W\ op_1\ op_2\ \mathsf{eq\_op}) \Longrightarrow \mathsf{eq\_op}\ W\ op_1\ op_2
\end{array}
$$

Here, rel_prod relates the elements of two pairs point-wise using the given relations. The coinduction principle of eq_op is:

$$
\begin{array}{l}
(\bigwedge W\ op_1\ op_2\ ev.\ R\ \ W\ op_1\ op_2 \Longrightarrow \mathsf{eq\_op\_lifted}\ ev\ W\ op_1\ op_2\ R) \Longrightarrow \\
R\ \ W\ op_1\ op_2 \Longrightarrow \mathsf{eq\_op}\ \ W\ op_1\ op_2
\end{array} \tag{25}
$$

The relation eq_op is a reasonable equivalence of operators: two operators related by eq_op generate the same outputs via produce when applied to the same monotone input:

$$\mathsf{mono}\ lxs\ W \Longrightarrow \mathsf{eq\_op}\ \ W\ op_1\ op_2 \Longrightarrow \mathsf{produce}\ op_1\ \ lxs = \mathsf{produce}\ op_2\ \ lxs \tag{26}$$

This fact follows by the coinduction up to congruence (1). Finally, we prove that our two histogram implementations are related using the coinduction principle for eq_op (25):

$$\forall t \in \mathsf{set\_t}\ abs.\ \exists wm \in W.\ t \leq wm \Longrightarrow \forall t \in \mathsf{set\_t}\ tds.\ \forall wm \in W\ .\ t > wm \Longrightarrow \\ \mathsf{eq\_op}\ W\ (\mathsf{ihist\_op}'\ (\mathsf{mset}\ (\mathsf{map}\ \mathsf{snd}\ abs))\ tds)\ (\mathsf{ihist\_op}\ tds\ abs) \tag{27}$$

Combining this fact with the soundness of eq_op (26) implies our desired property (24).

Our incremental histogram operators are executable using the lazy evaluation library by Lochbihler and Stoop [25]. In particular, our definitions do not use any non-executable constants such as quantifiers over infinite domains. Furthermore, produce has code equations, which enables code generation [14]. Naturally, when working with infinite input lazy lists, one can only observe finite prefixes of the output in finite time, e.g., by using ltake_DT.

5.2 Join

Our second case study illustrates how multiple disjoint time-aware data streams can be represented by a single one, which allows us to define operators with multiple inputs. The operator, defined in Figure 9, joins data items using a given $join :: {'a} \Rightarrow {'a} \Rightarrow {'b}\ option$ function, and uses the sum type ${'t} + {'t}$ as its time-stamp type: an input with time-stamp Inl t_1 is interpreted as coming the first data stream, whereas time-stamp Inr t_2 originates from the second data stream. We use the point-wise partial order on sum types as the order for our definition. That is, Inl t_1 is not related to any Inr t_2, and Inl $t_1 \leq$ Inl t_2 iff $t_1 \leq t_2$ (and similarly for Inr).

Once again, we use the incr_batch_op operator, so that we only need to define how to compute the results from the accumulated batches. In Figure 9, we apply join_list to each accumulated batch, which combines Inl and Inr tuples when they share the same time-stamp, and apply the provided *join* function to all possible combinations. Each joined result is output as an individual data item by the flatten operator flatten_op, which transforms batches into individual data items assigning the time-stamp t from DT t b to each element in the batch b. Finally, we remove the sum type from the output time-stamps using the union operator union_op, which outputs watermarks as soon they are identified as *producible*, meaning that a greater or equal watermark on the opposite side was already consumed. This identification is relevant for the preservation of mono, as it is only safe to output a watermark wm after the data on both sides is completed for it. The operator union_op has two buffers: the first keeps track of watermarks not identified as producible yet; the second denotes the maximal antichain of seen watermarks.

Figure 10 shows the four correctness properties of join_op with empty initial buffers to simplify the involved assumptions. The first two (soundness and completeness) express, respectively, that each joined item is the result of joining elements from different sides of the input data stream, and that every pair of items that can be joined will generate an item in the output. Both proofs follow similarly to the soundness (22) and completeness (23) of ihist_op. Preservation of prod for join_op assumes all incoming watermarks to be producible. Under this assumption, all watermarks will be output. Our proofs rely on four correctness properties of union_op (omitted), that resemble in terms of additional assumptions closely the properties of join_op.

6 Conclusion

We used the Isabelle/HOL proof assistant to model (possibly non-terminating) time-aware data stream processing. We identified two essential building block operators for batching and incremental computations, which we reused in two case studies. Moreover, we established the correctness for our operators, combining induction and coinduction. The correctness of composed operators follows compositionally from the building block operators' correctness.

corec flatten_op **where**
  flatten_op = Logic ($\lambda ev.$ case ev of WM wm ⇒ (flatten_op, [WM wm])
    | DT t d ⇒ (flatten_op, map (DT t) d)) [#]
definition join_list **where**
  join_list $join$ st xs = (let t = case_sum id id st;
   $lefts$ = map_filter ($\lambda(v, d)$. case v of Inr _ ⇒ None
    | Inl t' ⇒ if $t = t'$ then Some d else None) xs;
   $rights$ = map_filter ($\lambda(v, d)$. case v of Inl _ ⇒ None
    | Inr t' ⇒ if $t = t'$ then Some d else None) xs in
   concat (map ($\lambda d1$. map_filter (λd_2 . $join$ d_1 d_2) $rights$) $lefts$))
definition producible wm MA = ($\exists wm' \in MA.$ $wm \leq$ case_sum Inr Inl wm')
corec union_op **where** union_op wms MA = Logic (λ ev. case ev of
    DT t d ⇒ (union_op wms MA, [DT (case_sum id id t) d])
   | WM wm ⇒ let MA' = maximal_antichain_set (insert wm MA) ;
     $prds$ = {wm' ∈ set (wm # wms). producible wm' MA'};
     (union_op [wm' ← wm # wms. $wm' \notin prds$] MA',
      map (case_sum WM WM) [wm' ← wm # wms. $wm' \in prds$])) [#]
definition join_op tds abs wms MA $join$ = comp_op (comp_op (incr_batch_op tds abs)
  (comp_op (map_op (join_list $join$)) flatten_op)) (union_op wms MA)

Figure 9 Flatten, union, and join operators

mono lxs $W \Longrightarrow$ DT t d ∈ lset (produce (join_op [] [] [] {} $join$) lxs) $\Longrightarrow$
$\exists d_1\ d_2.\ join\ d_1\ d_2$ = Some $d \wedge d_1 \in$ set (DTs lxs (Inl t)) $\wedge$ $d_2 \in$ set (DTs lxs (Inr t)) (28)

mono lxs $W \Longrightarrow$ prod $lxs \Longrightarrow$ DT (Inl t) $d_1 \in$ lset $lxs \Longrightarrow$ DT (Inr t) $d_2 \in$ lset $lxs \Longrightarrow$
$join\ d_1\ d_2$ = Some $d \Longrightarrow$ DT t d ∈ lset (produce (join_op [] [] [] {} $join$) lxs) (29)

mono lxs {} $\Longrightarrow$ mono (produce (join_op [] [] [] {} $join$)) {} (30)

prod $lxs \Longrightarrow \forall wm \in$ vimage WM (lset lxs). producible wm (vimage WM (lset lxs)) $\Longrightarrow$
prod (produce (join_op [] [] [] {} $join$)) (31)

Figure 10 Correctness properties of join_op

We further introduced a reusable generalization technique using the skip_op operator that allows (co)inductive reasoning about elements at arbitrary positions.

We benefited from Isabelle's infrastructure for coinductive predicates, codatatypes, and corecursive functions, especially the support for friendly corecursion and monadic recursion and associated reasoning principles. A point for future work for Isabelle's developers could be to automatically derive coinduction up to congruence principles for coinductive predicates such as our manually derived principles (8) and (9). Our formalization amounts to around 17 000 lines of definitions and proofs. Of these, the heavy lifting happens for basic libraries (6 000) and reusable operators (9 000) with batch_op being the main culprit (5 000). In contrast, compositional reasoning in our case studies (Section 5) benefits from this groundwork (2 000).

As future work, we want to use partially-ordered time-stamps to introduce a feedback loop operator as in the Timely Dataflow stream processing framework [29, 31, 32]. Moreover, we currently do not support parallelism. A long-term goal to extend operators with a notion of workers they run on, which will enable us to distribute the input stream across workers as in Timely Dataflow and reason about the correctness of the resulting distributed streaming computation. Our formalization is executable, but it is not efficient because relies on the code extraction to purely functional languages. We plan to connect our work to the Isabelle-LLVM refinement framework [20] to obtain efficient executable operators.

References

1. The Isabelle/HOL formalization of the results. `https://github.com/rafaelcgs10/verified_stream_processing`. Accessed: 2024-03-18.
2. Martín Abadi, Frank McSherry, and Gordon D. Plotkin. Foundations of differential dataflow. In Andrew M. Pitts, editor, *Foundations of Software Science and Computation Structures - 18th International Conference, FoSSaCS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9034 of *Lecture Notes in Computer Science*, pages 71–83. Springer, 2015. `doi:10.1007/978-3-662-46678-0\_5`.
3. Andreas Abel and Brigitte Pientka. Well-founded recursion with copatterns and sized types. *J. Funct. Program.*, 26:e2, 2016. `doi:10.1017/S0956796816000022`.
4. 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. *Proc. VLDB Endow.*, 8(12):1792–1803, 2015. `doi:10.14778/2824032.2824076`.
5. Ahmed Alquraan, Hatem Takruri, Mohammed Alfatafta, and Samer Al-Kiswany. An analysis of network-partitioning failures in cloud systems. In Andrea C. Arpaci-Dusseau and Geoff Voelker, editors, *OSDI 2018*, pages 51–68. USENIX Association, 2018.
6. Edmon Begoli, Tyler Akidau, Slava Chernyak, Fabian Hueske, Kathryn Knight, Kenneth L. Knowles, Daniel Mills, and Dan Sotolongo. Watermarks in stream processing systems: Semantics and comparative analysis of apache flink and google cloud dataflow. *Proc. VLDB Endow.*, 14(12):3135–3147, 2021. `doi:10.14778/3476311.3476389`.
7. Julian Biendarra, Jasmin Christian Blanchette, Aymeric Bouzy, Martin Desharnais, Mathias Fleury, Johannes Hölzl, Ondrej Kuncar, Andreas Lochbihler, Fabian Meier, Lorenz Panny, Andrei Popescu, Christian Sternagel, René Thiemann, and Dmitriy Traytel. Foundational (co)datatypes and (co)recursion for higher-order logic. In Clare Dixon and Marcelo Finger, editors, *FroCoS 2017*, volume 10483 of *LNCS*, pages 3–21. Springer, 2017. `doi:10.1007/978-3-319-66167-4\_1`.
8. Jasmin Christian Blanchette, Aymeric Bouzy, Andreas Lochbihler, Andrei Popescu, and Dmitriy Traytel. Friends with benefits - implementing corecursion in foundational proof assistants. In Hongseok Yang, editor, *ESOP 2017*, volume 10201 of *LNCS*, pages 111–140. Springer, 2017. `doi:10.1007/978-3-662-54434-1\_5`.
9. Jasmin Christian Blanchette, Johannes Hölzl, Andreas Lochbihler, Lorenz Panny, Andrei Popescu, and Dmitriy Traytel. Truly modular (co)datatypes for Isabelle/HOL. In Gerwin Klein and Ruben Gamboa, editors, *ITP 2014*, volume 8558 of *LNCS*, pages 93–110. Springer, 2014. `doi:10.1007/978-3-319-08970-6\_7`.
10. Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink™: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38(4):28–38, 2015. URL: `http://sites.computer.org/debull/A15dec/p28.pdf`.
11. Nils Anders Danielsson and Thorsten Altenkirch. Subtyping, declaratively. In Claude Bolduc, Josée Desharnais, and Béchir Ktari, editors, *MPC 2010*, volume 6120 of *LNCS*, pages 100–118. Springer, 2010. `doi:10.1007/978-3-642-13321-3\_8`.
12. Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In Eric A. Brewer and Peter Chen, editors, *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, pages 137–150. USENIX Association, 2004. URL: `http://www.usenix.org/events/osdi04/tech/dean.html`.
13. Neil Ghani, Peter G. Hancock, and Dirk Pattinson. Representations of stream processors using nested fixed points. *Log. Methods Comput. Sci.*, 5(3), 2009.

14 Florian Haftmann and Tobias Nipkow. Code generation via higher-order rewrite systems. In Matthias Blume, Naoki Kobayashi, and Germán Vidal, editors, *FLOPS 2010*, volume 6009 of *LNCS*, pages 103–117. Springer, 2010. doi:10.1007/978-3-642-12251-4_9.

15 Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. Actris: session-type based reasoning in separation logic. *Proc. ACM Program. Lang.*, 4(POPL):6:1–6:30, 2020. doi:10.1145/3371074.

16 Gilles Kahn. The semantics of a simple language for parallel programming. In Jack L. Rosenfeld, editor, *Information Processing, Proceedings of the 6th IFIP Congress 1974, Stockholm, Sweden, August 5-10, 1974*, pages 471–475. North-Holland, 1974.

17 Kyle Kingsbury. Jepsen: Analyses. accessed: Oct 8, 2023. URL: https://jepsen.io/analyses.

18 Martin Kleppmann and Jay Kreps. Kafka, samza and the unix philosophy of distributed data. *IEEE Data Eng. Bull.*, 38(4):4–14, 2015. URL: http://sites.computer.org/debull/A15dec/p4.pdf.

19 Alexander Krauss. Recursive definitions of monadic functions. In Ekaterina Komendantskaya, Ana Bove, and Milad Niqui, editors, *Partiality and Recursion in Interactive Theorem Provers, PAR@ITP 2010*, volume 5 of *EPiC Series*, pages 1–13. EasyChair, 2010. doi:10.29007/1mdt.

20 Peter Lammich. Refinement of parallel algorithms down to LLVM. In June Andronick and Leonardo de Moura, editors, *13th International Conference on Interactive Theorem Proving, ITP 2022, August 7-10, 2022, Haifa, Israel*, volume 237 of *LIPIcs*, pages 24:1–24:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.ITP.2022.24.

21 Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. TaxDC: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. In Tom Conte and Yuanyuan Zhou, editors, *ASPLOS 2016*, pages 517–530. ACM, 2016.

22 Haopeng Liu, Guangpu Li, Jeffrey F. Lukman, Jiaxin Li, Shan Lu, Haryadi S. Gunawi, and Chen Tian. DCatch: Automatically detecting distributed concurrency bugs in cloud systems. In Yunji Chen, Olivier Temam, and John Carter, editors, *ASPLOS 2017*, pages 677–691. ACM, 2017.

23 Haopeng Liu, Xu Wang, Guangpu Li, Shan Lu, Feng Ye, and Chen Tian. FCatch: Automatically detecting time-of-fault bugs in cloud systems. In Xipeng Shen, James Tuck, Ricardo Bianchini, and Vivek Sarkar, editors, *ASPLOS 2018*, pages 419–431. ACM, 2018.

24 Andreas Lochbihler and Johannes Hölzl. Recursive functions on lazy lists via domains and topologies. In Gerwin Klein and Ruben Gamboa, editors, *ITP 2014*, volume 8558 of *LNCS*, pages 341–357. Springer, 2014. doi:10.1007/978-3-319-08970-6_22.

25 Andreas Lochbihler and Pascal Stoop. Lazy algebraic types in Isabelle/HOL, 2018.

26 Rupak Majumdar and Filip Niksic. Why is random testing effective for partition tolerance bugs? *PACMPL*, 2(POPL):46:1–46:24, 2018.

27 Konstantinos Mamouras. Semantic foundations for deterministic dataflow and stream processing. In Peter Müller, editor, *ESOP 2020*, volume 12075 of *LNCS*, pages 394–427. Springer, 2020. doi:10.1007/978-3-030-44914-8_15.

28 Frank McSherry. Github: Differential dataflow. URL: https://github.com/timelydataflow/differential-dataflow/.

29 Frank McSherry. Github: Timely dataflow. URL: https://github.com/TimelyDataflow/timely-dataflow/.

30 Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. Differential dataflow. In *Sixth Biennial Conference on Innovative Data Systems Research, CIDR 2013, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*. www.cidrdb.org, 2013.

31 Derek Gordon Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: a timely dataflow system. In Michael Kaminsky and Mike Dahlin, editors, *SOSP 2013*, pages 439–455. ACM, 2013. doi:10.1145/2517349.2522738.

32 Derek Gordon Murray, Frank McSherry, Michael Isard, Rebecca Isaacs, Paul Barham, and Martín Abadi. Incremental, iterative data processing with timely dataflow. *Commun. ACM*, 59(10):75–83, 2016. doi:10.1145/2983551.

33 Lawrence C. Paulson. *A Fixedpoint Approach to (Co)Inductive and (Co)Datatype Definitions*, page 187–211. MIT Press, Cambridge, MA, USA, 2000.

34 Robert Sandner and Olaf Müller. Theorem prover support for the refinement of functions. In Ed Brinksma, editor, *TACAS 1997*, volume 1217 of *LNCS*, pages 351–365. Springer, 1997. doi:10.1007/BFb0035399.

35 Maria Spichkova. *Specification and seamless verification of embedded real-time systems: FOCUS on Isabelle*. PhD thesis, Technical University Munich, Germany, 2007. URL: http://mediatum.ub.tum.de/doc/620981/document.pdf.

36 Caleb Stanford, Konstantinos Kallas, and Rajeev Alur. Correctness in stream processing: Challenges and opportunities. In *CIDR 2022*. www.cidrdb.org, 2022.

37 Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay Jain, and Michael Stumm. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In Jason Flinn and Hank Levy, editors, *OSDI 2014*, pages 249–265. USENIX Association, 2014.

38 Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: fault-tolerant streaming computation at scale. In Michael Kaminsky and Mike Dahlin, editors, *SOSP 2013*, pages 423–438. ACM, 2013. doi:10.1145/2517349.2522737.