Verified Time-Aware Stream Processing

Rafael Castro G. Silva

rasi@di.ku.dk

Department of Computer Science
University of Copenhagen

02/11/2023

What is this PhD/Status seminar about?

- Distributed Systems
  - Stream processing frameworks
    - Dataflow models
      - Time-Aware Computations
- Formal Methods
  - Verification using proof assistants
    - Isabelle proofs
      - Verified + executable + efficient code
- Formalization of Time-Aware Stream Processing

What is this PhD/Status seminar about?

- Distributed Systems
  - Stream processing frameworks
    - Dataflow models
      - Time-Aware Computations
- Formal Methods
  - Verification using proof assistants
    - Isabelle proofs
      - Verified + executable + efficient code
- Formalization of Time-Aware Stream Processing

What is this PhD/Status seminar about?

- Distributed Systems
    - Stream processing frameworks
        - Dataflow models
            - Time-Aware Computations
- Formal Methods
    - Verification using proof assistants
        - Isabelle proofs
            - Verified + executable + efficient code
- Formalization of Time-Aware Stream Processing

Contents

Introduction

Stream Processing

- Stream Processing: Abstraction for processing data when the input is not completely presented in the begging of the computation
- Dataflow Model:
  - Directed graph of interconnected operators that perform event-wise transformations
  - Examples: Apache Flink, Apache Samza, Apache Spark, Google Cloud Dataflow, and Timely Dataflow
  - Highly Parallel

- Time-Aware Computations
  - Timestamps: Metadata associating the data with some data collection
  - Watermarks: Metadata indicating the completion of a data collection

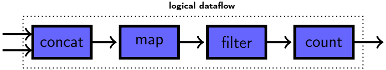
Stream Processing

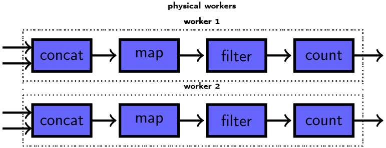
- Stream Processing: Abstraction for processing data when the input is not completely presented in the begging of the computation
- Dataflow Model:
  - Directed graph of interconnected operators that perform event-wise transformations
  - Examples: Apache Flink, Apache Samza, Apache Spark, Google Cloud Dataflow, and Timely Dataflow
  - Highly Parallel

- Time-Aware Computations
  - Timestamps: Metadata associating the data with some data collection
  - Watermarks: Metadata indicating the completion of a data collection

Stream Processing

- Stream Processing: Abstraction for processing data when the input is not completely presented in the begging of the computation
- Dataflow Model:
  - Directed graph of interconnected operators that perform event-wise transformations
  - Examples: Apache Flink, Apache Samza, Apache Spark, Google Cloud Dataflow, and Timely Dataflow
  - Highly Parallel





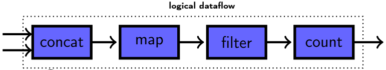
- Time-Aware Computations
  - Timestamps: Metadata associating the data with some data collection
  - Watermarks: Metadata indicating the completion of a data collection

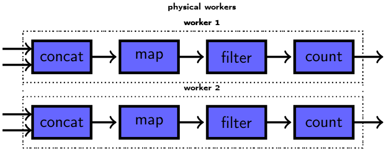
Stream Processing

- Stream Processing: Abstraction for processing data when the input is not completely presented in the begging of the computation
- Dataflow Model:
  - Directed graph of interconnected operators that perform event-wise transformations
  - Examples: Apache Flink, Apache Samza, Apache Spark, Google Cloud Dataflow, and Timely Dataflow

Cloud DataFlow Flink

  - Highly Parallel





- Time-Aware Computations
  - Timestamps: Metadata associating the data with some data collection
  - Watermarks: Metadata indicating the completion of a data collection

Preliminaries

Isabelle/HOL

- Classical higher-order logic (HOL): Simple Typed Lambda Calculus + (Hilbert) axiom of choice + axiom of infinity + rank-1 polymorphism
- Isabelle: A generic proof assistant
- Isabelle/HOL: Isabelle's flavor of HOL
- All functions in Isabelle/HOL must be total

Isabelle/HOL

- Classical higher-order logic (HOL): Simple Typed Lambda Calculus + (Hilbert) axiom of choice + axiom of infinity + rank-1 polymorphism
- Isabelle: A generic proof assistant

- Isabelle/HOL: Isabelle's flavor of HOL
- All functions in Isabelle/HOL must be total

Isabelle/HOL: (Co)datatypes

- Datatypes and Codatatypes

  codatatype (lset: $'a$) *llist* = lnull: LNil | LCons (lhd: $'a$) (ltl: $'a$ *llist*)
  for map: lmap **where** ltl LNil = LNil

- Examples:
  - LNil
  - LCons *1* (LCons *2* (LCons *3* LNil))
  - LCons *0* (LCons *0* (LCons *0* (...)))
- Induction principle assuming membership in the lazy list
- Coinductive principle for lazy list equality:
  - Show that there is a pair of goggles that makes them to look the same, which implies that:
    - The first lazy list is empty iff second is
    - They have the same head
    - Their tail looks the same

Isabelle/HOL: (Co)datatypes

- Datatypes and Codatatypes

  codatatype (lset: $'a$) *llist* = lnull: LNil | LCons (lhd: $'a$) (ltl: $'a$ *llist*)
  for map: lmap **where** ltl LNil = LNil

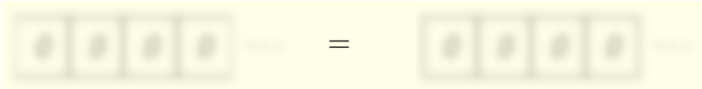
- Examples:
  - LNil
  - LCons *1* (LCons *2* (LCons *3* LNil))
  - LCons *0* (LCons *0* (LCons *0* (...)))
- Induction principle assuming membership in the lazy list
- Coinductive principle for lazy list equality:
  - Show that there is a pair of goggles that makes them to look the same, which implies that:
    - The first lazy list is empty iff second is
    - They have the same head
    - Their tail looks the same

Isabelle/HOL: (Co)datatypes

- Datatypes and Codatatypes

  codatatype (lset: $'a$) *llist* = lnull: LNil | LCons (lhd: $'a$) (ltl: $'a$ *llist*)
  for map: lmap **where** ltl LNil = LNil

- Examples:
  - LNil
  - LCons *1* (LCons *2* (LCons *3* LNil))
  - LCons *0* (LCons *0* (LCons *0* (...)))
- Induction principle assuming membership in the lazy list
- Coinductive principle for lazy list equality:
  - Show that there is a pair of goggles that makes them to look the same, which implies that:
    - The first lazy list is empty iff second is
    - They have the same head
    - Their tail looks the same

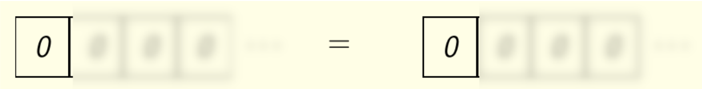


Isabelle/HOL: (Co)datatypes

- Datatypes and Codatatypes

  codatatype (lset: *'a*) *llist* = lnull: LNil | LCons (lhd: *'a*) (ltl: *'a llist*)
  for map: lmap **where** ltl LNil = LNil

- Examples:
  - LNil
  - LCons *1* (LCons *2* (LCons *3* LNil))
  - LCons *0* (LCons *0* (LCons *0* (...)))
- Induction principle assuming membership in the lazy list
- Coinductive principle for lazy list equality:
  - Show that there is a pair of goggles that makes them to look the same, which must imply that:
    - The first lazy list is empty iff second is
    - They have the same head
    - Their tail looks the same



Isabelle/HOL: (Co)datatypes

- Datatypes and Codatatypes

  codatatype (lset: *'a*) *llist* = lnull: LNil | LCons (lhd: *'a*) (ltl: *'a llist*)
  for map: lmap **where** ltl LNil = LNil

- Examples:
  - LNil
  - LCons *1* (LCons *2* (LCons *3* LNil))
  - LCons *0* (LCons *0* (LCons *0* (...)))
- Induction principle assuming membership in the lazy list
- Coinductive principle for lazy list equality:
  - Show that there is a pair of goggles that makes them to look the same, which implies that:
    - The first lazy list is empty iff second is
    - They have the same head
    - Their tail looks the same

$$\boxed{0}\boxed{0}\boxed{0}\boxed{0}\cdots \quad = \quad \boxed{0}\boxed{0}\boxed{0}\boxed{0}\cdots$$

Isabelle/HOL: Recursion and While Combinator

- Recursion

  $\textbf{fun}\ \mathsf{lshift} :: {'a}\ \mathit{list} \Rightarrow {'a}\ \mathit{llist} \Rightarrow {'a}\ \mathit{llist}\ (\mathit{infixr}\ @@\ 65)\ \textbf{where}$
  $\quad \mathsf{lshift}\ []\ \mathit{lxs} = \mathit{lxs}$
  $\mid \mathsf{lshift}\ (x \# \mathit{xs})\ \mathit{lxs} = \mathsf{LCons}\ x\ (\mathsf{lshift}\ \mathit{xs}\ \mathit{lxs})$

- While Combinator

  $\textbf{definition}\ \mathsf{while\_option} :: ({'a} \Rightarrow \mathit{bool}) \Rightarrow ({'a} \Rightarrow {'a}) \Rightarrow {'a} \Rightarrow {'a}\ \mathit{option}\ \textbf{where}$
  $\mathsf{while\_option}\ b\ c\ s = \ldots$

- While rule for invariant reasoning (hoare-style):
  - There is something that holds before a step; that thing still holds after the step

Isabelle/HOL: Corecursion and Friends

- Corecursion is like recursion, but instead of always eventually reducing an argument it always eventually produces something
- Corec:

$\textbf{corec}$ lapp $:: {}'a\ \mathit{llist} \Rightarrow {}'a\ \mathit{llist} \Rightarrow {}'a\ \mathit{llist}$ $\textbf{where}$
  lapp $\mathit{lxs}\ \mathit{lys}$ = case lxs of LNil $\Rightarrow \mathit{lys}$ | LCons $x\ \mathit{lxs}' \Rightarrow$ LCons x (lapp $\mathit{lxs}'\ \mathit{lys}$)

- Friendly function
  - Preserves productivity: it may consume at most one constructor to produce one constructor.

$\textbf{friend\_of\_corec}$ lshift $\textbf{where}$
  xs @@ lxs = (case xs of
    [] $\Rightarrow$ (case lxs of LNil $\Rightarrow$ LNil | LCons $x\ \mathit{lxs}' \Rightarrow$ LCons $x\ \mathit{lxs}'$)
  | $x\#\mathit{xs}' \Rightarrow$ LCons x (xs' @@ lxs))
  $\textbf{by}$ (*auto split: list.splits llist.splits*) (*transfer_prover*)

lconcat lxs = case lxs of LNil $\Rightarrow$ LNil | LCons $\mathit{xs}\ \mathit{lxs}' \Rightarrow$ lshift xs (lconcat lxs')

- Coinduction up to congruence: Coinduction for Lazy list equality can be extended to compare an entire finite prefix through a congruence relation

Isabelle/HOL: Corecursion and Friends

- Corecursion is like recursion, but instead of always eventually reducing an argument it always eventually produces something
- Corec:

$$\textbf{corec}\ \mathsf{lapp} :: {'a}\ \mathit{llist} \Rightarrow {'a}\ \mathit{llist} \Rightarrow {'a}\ \mathit{llist}\ \textbf{where}$$
$$\mathsf{lapp}\ \mathit{lxs}\ \mathit{lys} = \mathsf{case}\ \mathit{lxs}\ \mathsf{of}\ \mathsf{LNil} \Rightarrow \mathit{lys} \mid \mathsf{LCons}\ x\ \mathit{lxs'} \Rightarrow \mathsf{LCons}\ x\ (\mathsf{lapp}\ \mathit{lxs'}\ \mathit{lys})$$

- Friendly function
  - Preserves productivity: it may consume at most one constructor to produce one constructor.

$$\textbf{friend\_of\_corec}\ \mathsf{lshift}\ \textbf{where}$$
$$\mathit{xs}\ @@\ \mathit{lxs} = (\mathsf{case}\ \mathit{xs}\ \mathsf{of}$$
$$[] \Rightarrow (\mathsf{case}\ \mathit{lxs}\ \mathsf{of}\ \mathsf{LNil} \Rightarrow \mathsf{LNil} \mid \mathsf{LCons}\ x\ \mathit{lxs'} \Rightarrow \mathsf{LCons}\ x\ \mathit{lxs'})$$
$$\mid x\#\mathit{xs'} \Rightarrow \mathsf{LCons}\ x\ (\mathit{xs'}\ @@\ \mathit{lxs}))$$
$$\textbf{by}\ (\mathit{auto\ split{:}\ list.splits\ llist.splits})\ (\mathit{transfer\_prover})$$

$$\mathsf{lconcat}\ \mathit{lxs} = \mathsf{case}\ \mathit{lxs}\ \mathsf{of}\ \mathsf{LNil} \Rightarrow \mathsf{LNil} \mid \mathsf{LCons}\ \mathit{xs}\ \mathit{lxs'} \Rightarrow \mathsf{lshift}\ \mathit{xs}\ (\mathsf{lconcat}\ \mathit{lxs'})$$

- Coinduction up to congruence: Coinduction for Lazy list equality can be extended to compare an entire finite prefix through a congruence relation

Isabelle/HOL: Corecursion and Friends

- Corecursion is like recursion, but instead of always eventually reducing an argument it always eventually produces something
- Corec:

corec lapp :: $'a$ *llist* $\Rightarrow$ $'a$ *llist* $\Rightarrow$ $'a$ *llist* **where**
  lapp *lxs lys* = case *lxs* of LNil $\Rightarrow$ *lys* | LCons *x lxs'* $\Rightarrow$ LCons *x* (lapp *lxs' lys*)

- Friendly function
  - Preserves productivity: it may consume at most one constructor to produce one constructor.

friend_of_corec lshift **where**
  xs @@ *lxs* = (case *xs* of
    [] $\Rightarrow$ (case *lxs* of LNil $\Rightarrow$ LNil | LCons *x lxs'* $\Rightarrow$ LCons *x lxs'*)
  | *x*#*xs'* $\Rightarrow$ LCons *x* (*xs'* @@ *lxs*))
  by (*auto split: list.splits llist.splits*) (*transfer_prover*)

lconcat *lxs* = case *lxs* of LNil $\Rightarrow$ LNil | LCons *xs lxs'* $\Rightarrow$ lshift *xs* (lconcat *lxs'*)

- Coinduction up to congruence: Coinduction for Lazy list equality can be extended to compare an entire finite prefix through a congruence relation

Isabelle/HOL: Corecursion and Friends

- Corecursion is like recursion, but instead of always eventually reducing an argument it always eventually produces something
- Corec:

corec lapp :: $'a$ *llist* $\Rightarrow$ $'a$ *llist* $\Rightarrow$ $'a$ *llist* **where**
  lapp *lxs lys* = case *lxs* of LNil $\Rightarrow$ *lys* | LCons *x lxs'* $\Rightarrow$ LCons *x* (lapp *lxs' lys*)

- Friendly function
  - Preserves productivity: it may consume at most one constructor to produce one constructor.

friend_of_corec lshift **where**
  xs @@ *lxs* = (case *xs* of
    [] $\Rightarrow$ (case *lxs* of LNil $\Rightarrow$ LNil | LCons *x lxs'* $\Rightarrow$ LCons *x lxs'*)
  | *x*#*xs'* $\Rightarrow$ LCons *x* (*xs'* @@ *lxs*))
  by (*auto split: list.splits llist.splits*) (*transfer_prover*)

lconcat *lxs* = case *lxs* of LNil $\Rightarrow$ LNil | LCons *xs lxs'* $\Rightarrow$ lshift *xs* (lconcat *lxs'*)

- Coinduction up to congruence: Coinduction for Lazy list equality can be extended to compare an entire finite prefix through a congruence relation

Isabelle/HOL: (Co)inductive Predicates

- Inductive predicate
  - Finite number of introduction rule applications

inductive in_llist $:: 'a \Rightarrow 'a\ llist \Rightarrow bool$ **where**
  In_llist: in_llist x (LCons x lxs)
| Next_llist: in_llist x lxs $\Rightarrow$ in_llist x (LCons y lxs)

in_llist 2 (LCons 1 (LCons (2 (...))))

- Coinductive predicate
  - Infinite number of introduction rule applications

coinductive lprefix $:: 'a\ llist \Rightarrow 'a\ llist \Rightarrow bool$ **where**
  LNil_lprefix: lprefix LNil lys
| LCons_lprefix: lprefix lxs lys $\Rightarrow$ lprefix (LCons x lxs) (LCons x lys)

lprefix (LCons 1 (LCons (2 (...)))) (LCons 1 (LCons (2 (...))))

- Coinduction principle
- But not coinduction up to congruence for free

Isabelle/HOL: (Co)inductive Predicates

- Inductive predicate
  - Finite number of introduction rule applications

$$
\begin{array}{l}
\textbf{inductive}\ \mathsf{in\_llist} :: {'a} \Rightarrow {'a}\ \mathit{llist} \Rightarrow \mathit{bool}\ \textbf{where} \\
\quad \mathsf{In\_llist}\text{: } \mathsf{in\_llist}\ x\ (\mathsf{LCons}\ x\ \mathit{lxs}) \\
\ \mid \mathsf{Next\_llist}\text{: } \mathsf{in\_llist}\ x\ \mathit{lxs} \Rightarrow \mathsf{in\_llist}\ x\ (\mathsf{LCons}\ y\ \mathit{lxs}) \\
\\
\mathsf{in\_llist}\ \mathit{2}\ (\mathsf{LCons}\ \mathit{1}\ (\mathsf{LCons}\ (\mathit{2}\ (\ldots))))
\end{array}
$$

- Coinductive predicate
  - Infinite number of introduction rule applications

$$
\begin{array}{l}
\textbf{coinductive}\ \mathsf{lprefix} :: {'a}\ \mathit{llist} \Rightarrow {'a}\ \mathit{llist} \Rightarrow \mathit{bool}\ \textbf{where} \\
\quad \mathsf{LNil\_lprefix}\text{: } \mathsf{lprefix}\ \mathsf{LNil}\ \mathit{lys} \\
\ \mid \mathsf{LCons\_lprefix}\text{: } \mathsf{lprefix}\ \mathit{lxs}\ \mathit{lys} \Rightarrow \mathsf{lprefix}\ (\mathsf{LCons}\ x\ \mathit{lxs})\ (\mathsf{LCons}\ x\ \mathit{lys}) \\
\\
\mathsf{lprefix}\ (\mathsf{LCons}\ \mathit{1}\ (\mathsf{LCons}\ (\mathit{2}\ (\ldots))))\ (\mathsf{LCons}\ \mathit{1}\ (\mathsf{LCons}\ (\mathit{2}\ (\ldots))))
\end{array}
$$

- Coinduction principle
- But not coinduction up to congruence for free

Isabelle/HOL: (Co)inductive Predicates

- Inductive predicate
  - Finite number of introduction rule applications

inductive in_llist :: *'a* ⇒ *'a llist* ⇒ *bool* **where**
  In_llist: in_llist *x* (LCons *x* *lxs*)
| Next_llist: in_llist *x* *lxs* ⇒ in_llist *x* (LCons *y* *lxs*)

in_llist *2* (LCons *1* (LCons (*2* (...))))

- Coinductive predicate
  - Infinite number of introduction rule applications

coinductive lprefix :: *'a llist* ⇒ *'a llist* ⇒ *bool* **where**
  LNil_lprefix: lprefix LNil *lys*
| LCons_lprefix: lprefix *lxs* *lys* ⇒ lprefix (LCons *x* *lxs*) (LCons *x* *lys*)

lprefix (LCons *1* (LCons (*2* (...)))) (LCons *1* (LCons (*2* (...))))

- Coinduction principle
- But not coinduction up to congruence for free

Lazy Lists Processors

Operator formalization

- Operator as a codatatype
  - Taking $'i$ as the input type, and $'o$ as the output type:
    $\textbf{codatatype}\ ('o, 'i)\ op = \text{Logic}\ (\text{apply}\colon ('i \Rightarrow ('o, 'i)\ op \times 'o\ list))$
  - Infinite trees: applying the selector apply "walks" a branch of the tree

Operator formalization

- Operator as a codatatype
  - Taking $'i$ as the input type, and $'o$ as the output type:
    $\textbf{codatatype}\ ('o, 'i)\ op = \text{Logic}\ (\text{apply}: ('i \Rightarrow ('o, 'i)\ op \times 'o\ list))$
  - Infinite trees: applying the selector apply "walks" a branch of the tree

Execution formalization

- Produce function: applies the logic (co)recursively throughout a lazy list

$$
\begin{aligned}
&\textbf{definition}\ \mathsf{produce_1}'\ \mathit{op}\ \mathit{lxs} = \mathsf{while\_option}\\
&\quad (\lambda(\mathit{op}, \mathit{lxs}).\ \neg\ \mathsf{lnull}\ \mathit{lxs} \wedge \mathsf{snd}\ (\mathsf{apply}\ \mathit{op}\ (\mathsf{lhd}\ \mathit{lxs})) = [])\\
&\quad (\lambda(\mathit{op}, \mathit{lxs}).\ (\mathsf{fst}\ (\mathsf{apply}\ \mathit{op}\ (\mathsf{lhd}\ \mathit{lxs})), \mathsf{ltl}\ \mathit{lxs}))\ (\mathit{op}, \mathit{lxs})\\
&\textbf{definition}\ \mathsf{produce_1}\ \mathit{op}\ \mathit{lxs} =\\
&\quad (\mathtt{case}\ \mathsf{produce_1}'\ \mathit{op}\ \mathit{lxs}\ \mathtt{of}\ \mathsf{None} \Rightarrow \mathsf{None}\\
&\quad \mid \mathsf{Some}\ (\mathit{op}', \mathit{lxs}') \Rightarrow \mathtt{if}\ \mathsf{lnull}\ \mathit{lxs}'\ \mathtt{then}\ \mathsf{None}\ \mathtt{else}\\
&\qquad \mathtt{let}\ (\mathit{op}'', \mathit{out}) = \mathsf{apply}\ \mathit{op}'\ (\mathsf{lhd}\ \mathit{lxs}')\ \mathtt{in}\ \mathsf{Some}\ (\mathit{op}'', \mathsf{hd}\ \mathit{out}, \mathsf{tl}\ \mathit{out}, \mathsf{ltl}\ \mathit{lxs}'))\\
&\textbf{corec}\ \mathsf{produce}\ \textbf{where}\\
&\quad \mathsf{produce}\ \mathit{op}\ \mathit{lxs} = (\mathtt{case}\ \mathsf{produce_1}\ \mathit{op}\ \mathit{lxs}\ \mathtt{of}\ \mathsf{None} \Rightarrow \mathsf{LNil}\\
&\qquad \mid \mathsf{Some}\ (\mathit{op}', x, \mathit{xs}, \mathit{lxs}') \Rightarrow \mathsf{LCons}\ x\ (\mathit{xs}\ @@\ \mathsf{produce}\ \mathit{op}'\ \mathit{lxs}'))
\end{aligned}
$$

- $\mathsf{produce_1}$ has an induction principle based on the while invariant rule

Execution formalization

- Produce function: applies the logic (co)recursively throughout a lazy list

$$
\begin{aligned}
&\textbf{definition}\ \mathsf{produce_1}'\ \mathit{op}\ \mathit{lxs} = \mathsf{while\_option}\\
&\quad (\lambda(\mathit{op}, \mathit{lxs}).\ \neg\ \mathsf{lnull}\ \mathit{lxs} \wedge \mathsf{snd}\ (\mathsf{apply}\ \mathit{op}\ (\mathsf{lhd}\ \mathit{lxs})) = [])\\
&\quad (\lambda(\mathit{op}, \mathit{lxs}).\ (\mathsf{fst}\ (\mathsf{apply}\ \mathit{op}\ (\mathsf{lhd}\ \mathit{lxs})), \mathsf{ltl}\ \mathit{lxs}))\ (\mathit{op}, \mathit{lxs})\\
&\textbf{definition}\ \mathsf{produce_1}\ \mathit{op}\ \mathit{lxs} =\\
&\quad (\mathtt{case}\ \mathsf{produce_1}'\ \mathit{op}\ \mathit{lxs}\ \mathtt{of}\ \mathsf{None} \Rightarrow \mathsf{None}\\
&\quad \mid \mathsf{Some}\ (\mathit{op}', \mathit{lxs}') \Rightarrow \mathtt{if}\ \mathsf{lnull}\ \mathit{lxs}'\ \mathtt{then}\ \mathsf{None}\ \mathtt{else}\\
&\qquad \mathtt{let}\ (\mathit{op}'', \mathit{out}) = \mathsf{apply}\ \mathit{op}'\ (\mathsf{lhd}\ \mathit{lxs}')\ \mathtt{in}\ \mathsf{Some}\ (\mathit{op}'', \mathsf{hd}\ \mathit{out}, \mathsf{tl}\ \mathit{out}, \mathsf{ltl}\ \mathit{lxs}'))\\
&\textbf{corec}\ \mathsf{produce}\ \textbf{where}\\
&\quad \mathsf{produce}\ \mathit{op}\ \mathit{lxs} = (\mathtt{case}\ \mathsf{produce_1}\ \mathit{op}\ \mathit{lxs}\ \mathtt{of}\ \mathsf{None} \Rightarrow \mathsf{LNil}\\
&\qquad \mid \mathsf{Some}\ (\mathit{op}', x, \mathit{xs}, \mathit{lxs}') \Rightarrow \mathsf{LCons}\ x\ (\mathit{xs}\ @@\ \mathsf{produce}\ \mathit{op}'\ \mathit{lxs}'))
\end{aligned}
$$

- $\mathsf{produce_1}$ has an induction principle based on the while invariant rule

- Example:

corec count_op **where** count_op P n =
  Logic (λe. `if` P e `then` (count_op P $(n + 1)$, $[n{+}1]$) `else` (count_op P n, $[]$))

$$\mathit{stream}_2 = \boxed{0}\,\boxed{3}\,\boxed{3}\,\boxed{6}\,\boxed{24}$$

$$\text{produce (count\_op } \mathit{is\_even}\ 0\text{)}\ \mathit{stream}_2 = \boxed{1}\,\boxed{2}\,\boxed{3}$$

Sequential Composition

- Sequential composition: take the output of the first operator and give it as input to the second operator.

$$
\begin{array}{l}
\textbf{definition}\ \mathrm{fproduce}\ op\ xs = \mathrm{fold}\ (\lambda e\ (op, out).\\
\quad \mathtt{let}\ (op', out') = \mathrm{apply}\ op\ e\ \mathtt{in}\ (op', out\ @\ out'))\ xs\ (op, [])\\
\textbf{corec}\ \mathrm{comp\_op}\ \textbf{where}\\
\quad \mathrm{comp\_op}\ op_1\ op_2 = \mathrm{Logic}\ (\lambda ev.\\
\qquad \mathtt{let}\ (op_1', out) = \mathrm{apply}\ op_1\ ev;\ \ (op_2', out') = \mathrm{fproduce}\ op_2\ out\\
\qquad \mathtt{in}\ (\mathrm{comp\_op}\ op_1'\ op_2', out'))
\end{array}
$$

Sequential Composition: Correctness

- Correctness:

$$\text{produce (comp\_op } op_1 \ op_2) \ lxs = \text{produce } op_2 \ (\text{produce } op_1 \ lxs)$$

- Proof: coinduction principle for lazy list equality and produce$_1$ induction principle
  - Generalization: we must be able to reason about elements in arbitrary positions

```
corec skip_op where
  skip_op op n = Logic (λev. let (op', out) = apply op ev in
    if length out < n then (skip_op op' (n − length out), [])
    else (op', drop n out))
```

  - Correctness: Coinduction up to congruence for lazy list equality

Sequential Composition: Correctness

- Correctness:

  $\text{produce }(\text{comp\_op } op_1\ op_2)\ lxs = \text{produce } op_2\ (\text{produce } op_1\ lxs)$

- Proof: coinduction principle for lazy list equality and produce_1 induction principle
  - Generalization: we must be able to reason about elements in arbitrary positions

    $\textbf{corec}\ \text{skip\_op}\ \textbf{where}$
    $\quad \text{skip\_op}\ op\ n = \text{Logic}\ (\lambda ev.\ \texttt{let}\ (op', out) = \text{apply}\ op\ ev\ \texttt{in}$
    $\qquad \texttt{if}\ \text{length}\ out < n\ \texttt{then}\ (\text{skip\_op}\ op'\ (n - \text{length}\ out), [])$
    $\qquad \texttt{else}\ (op', \text{drop}\ n\ out))$

  - Correctness: Coinduction up to congruence for lazy list equality

Sequential Composition: Correctness

- Correctness:

$\text{produce}\ (\text{comp\_op}\ op_1\ op_2)\ lxs = \text{produce}\ op_2\ (\text{produce}\ op_1\ lxs)$

- Proof: coinduction principle for lazy list equality and produce_1 induction principle
  - Generalization: we must be able to reason about elements in arbitrary positions

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

  - Correctness: Coinduction up to congruence for lazy list equality

Sequential Composition: Correctness

- Correctness:

$$\text{produce } (\text{comp\_op } op_1 \ op_2) \ lxs = \text{produce } op_2 \ (\text{produce } op_1 \ lxs)$$

- Proof: coinduction principle for lazy list equality and produce$_1$ induction principle
  - Generalization: we must be able to reason about elements in arbitrary positions

$$\begin{array}{l}
\textbf{corec } \text{skip_op } \textbf{where} \\
\quad \text{skip_op } op \ n = \text{Logic } (\lambda ev. \ \texttt{let } (op', out) = \text{apply } op \ ev \ \texttt{in} \\
\qquad \texttt{if } \text{length } out < n \ \texttt{then } (\text{skip_op } op' \ (n - \text{length } out), []) \\
\qquad \texttt{else } (op', \text{drop } n \ out))
\end{array}$$

  - Correctness: Coinduction up to congruence for lazy list equality

Time-Aware Operators

Time-Aware Streams

- Time-Aware lazy lists

$$\textbf{datatype}\ ('t{::}order, 'd)\ \mathit{event} = \mathrm{DT}\ (\mathrm{tmp}{:}\ 't)\ (\mathrm{data}{:}\ 'd)\ |\ \mathrm{WM}\ (\mathrm{wmk}{:}\ 't)$$

- Generalization to partial orders
  - Cycles
  - Operators with multiple inputs

Time-Aware Streams

- Time-Aware lazy lists

$\textbf{datatype}\ ('t{::}order,\ 'd)\ event = \mathsf{DT}\ (\mathsf{tmp}{:}\ 't)\ (\mathsf{data}{:}\ 'd) \mid \mathsf{WM}\ (\mathsf{wmk}{:}\ 't)$

- Generalization to partial orders
  - Cycles
  - Operators with multiple inputs

- Monotone: watermarks do not go back in time

coinductive monotone $::$ $('t::order, 'd)$ *event llist* $\Rightarrow$ $'t$ set $\Rightarrow$ *bool* **where**
LNil: monotone LNil W
| LConsR: $(\forall wm' \in W.\ \neg\ wm' \geq wm) \longrightarrow$ monotone lxs $(\{wm\} \cup W) \longrightarrow$
monotone (LCons (WM wm) lxs) W
| LConsL: $(\forall wm \in W.\ \neg\ wm \geq t) \longrightarrow$ monotone lxs $W \longrightarrow$
monotone (LCons (DT t d) lxs) W

- Up to congruence coinduction principle
- Example:

$stream_3$ =

DT t_4 d	DT t_0 a	DT t_1 c	WM t_1	DT t_5 c
DT t_3 a	DT t_5 a	WM t_2	WM t_5	DT t_2 b

…

Monotone Time-Aware Streams

- Monotone: watermarks do not go back in time

  coinductive monotone :: $('t{::}order, 'd)$ *event llist* $\Rightarrow$ $'t$ set $\Rightarrow$ *bool* **where**
  LNil: monotone LNil W
  | LConsR: $(\forall wm' \in W.\ \neg\ wm' \geq wm) \longrightarrow$ monotone lxs $(\{wm\} \cup W) \longrightarrow$
  monotone (LCons (WM wm) lxs) W
  | LConsL: $(\forall wm \in W.\ \neg\ wm \geq t) \longrightarrow$ monotone lxs $W \longrightarrow$
  monotone (LCons (DT t d) lxs) W

- Up to congruence coinduction principle
- Example:

$stream_3 =$

DT t_4 d	DT t_0 a	DT t_1 c	WM t_1	DT t_5 c
DT t_3 a	DT t_5 a	WM t_2	WM t_5	DT t_2 b …

Monotone Time-Aware Streams

- Monotone: watermarks do not go back in time

$$
\begin{array}{l}
\textbf{coinductive}\ \text{monotone} :: ({}'t{::}order, {}'d)\ event\ llist \Rightarrow {}'t\ \text{set} \Rightarrow bool\ \textbf{where} \\
\quad \text{LNil}: \text{monotone LNil}\ W \\
\mid \text{LConsR}: (\forall wm' \in W.\ \neg\ wm' \geq wm) \longrightarrow \text{monotone}\ lxs\ (\{wm\} \cup W) \longrightarrow \\
\qquad \text{monotone (LCons (WM}\ wm)\ lxs)\ W \\
\mid \text{LConsL}: (\forall wm \in W.\ \neg\ wm \geq t) \longrightarrow \text{monotone}\ lxs\ W \longrightarrow \\
\qquad \text{monotone (LCons (DT}\ t\ d)\ lxs)\ W
\end{array}
$$

- Up to congruence coinduction principle
- Example:

$stream_3 =$

DT t_4 d	DT t_0 a	DT t_1 c	WM t_1	DT t_5 c
DT t_3 a	DT t_5 a	WM t_2	WM t_5	DT t_2 b

$\cdots$

$$
\begin{tikzcd}
C_{t_4}: \{\, d \,\} & & & C_{t_5}: \{\, a, c \,\} & & \\
C_{t_1}: \{\, c \,\} \arrow[u] & & C_{t_2}: \{\, b \,\} \arrow[ur] & & C_{t_3}: \{\, a \,\} \arrow[ul] & \\
& & C_{t_0}: \{\, a \,\} \arrow[ull] \arrow[u] \arrow[urr] & & &
\end{tikzcd}
$$

Productive Time-Aware Streams

- Productive: always eventually allows the production
  - Batching operators: accumulate data until its completion
  - Data is always eventually completed by some watermark

coinductive productive **where**
  LFinite: lfinite *lxs* $\longrightarrow$ productive *lxs*
| EnvWM: $\neg$ lfinite *lxs* $\longrightarrow$ ($\exists u \in$ vimage WM (lset *lxs*). $u \geq t$) $\longrightarrow$
  productive *lxs* $\longrightarrow$ productive (LCons (DT *t d*) *lxs*)
| *SkipWM:* $\neg$ lfinite *lxs* $\longrightarrow$ productive *lxs* $\longrightarrow$
  productive (LCons (WM *t*) *lxs*)

  - Up to congruence coinduction principle

Productive Time-Aware Streams

- Productive: always eventually allows the production
  - Batching operators: accumulate data until its completion
  - Data is always eventually completed by some watermark

coinductive productive **where**
  LFinite: lfinite *lxs* $\longrightarrow$ productive *lxs*
| EnvWM: $\neg$ lfinite *lxs* $\longrightarrow$ ($\exists u \in$ vimage WM (lset *lxs*). $u \geq t$) $\longrightarrow$
  productive *lxs* $\longrightarrow$ productive (LCons (DT t d) *lxs*)
| *SkipWM:* $\neg$ lfinite *lxs* $\longrightarrow$ productive *lxs* $\longrightarrow$
  productive (LCons (WM t) *lxs*)

  - Up to congruence coinduction principle

Productive Time-Aware Streams

- Productive: always eventually allows the production
  - Batching operators: accumulate data until its completion
  - Data is always eventually completed by some watermark

coinductive productive **where**
  LFinite: lfinite *lxs* $\longrightarrow$ productive *lxs*
| EnvWM: $\neg$ lfinite *lxs* $\longrightarrow$ $(\exists u \in$ vimage WM (lset *lxs*). $u \geq t)$ $\longrightarrow$
  productive *lxs* $\longrightarrow$ productive (LCons (DT *t* *d*) *lxs*)
| *SkipWM:* $\neg$ lfinite *lxs* $\longrightarrow$ productive *lxs* $\longrightarrow$
  productive (LCons (WM *t*) *lxs*)

  - Up to congruence coinduction principle

Productive Time-Aware Streams

- Productive: always eventually allows the production
  - Batching operators: accumulate data until its completion
  - Data is always eventually completed by some watermark

coinductive productive **where**
  LFinite: lfinite *lxs* $\longrightarrow$ productive *lxs*
| EnvWM: $\neg$ lfinite *lxs* $\longrightarrow$ $(\exists u \in$ vimage WM (lset *lxs*). $u \geq t)$ $\longrightarrow$
   productive *lxs* $\longrightarrow$ productive (LCons (DT t d) *lxs*)
| *SkipWM:* $\neg$ lfinite *lxs* $\longrightarrow$ productive *lxs* $\longrightarrow$
   productive (LCons (WM t) *lxs*)

  - Up to congruence coinduction principle

Building Blocks: Batch Operator

- Building Blocks: reusable operators
  - Batching and incremental computations
- batch_op : produces batches of accumulated data

$$
\begin{array}{l}
\textbf{corec}\ \text{batch\_op}\ \textbf{where} \\
\quad \text{batch\_op}\ \mathit{buf} = \text{Logic}\ (\lambda \mathit{ev}.\ \text{case}\ \mathit{ev}\ \text{of DT}\ t\ d \Rightarrow (\text{batch\_op}\ (\mathit{buf} @ [(t, d)]), [\,]) \\
\quad \mid \text{WM}\ \mathit{wm} \Rightarrow \texttt{if}\ \exists (t, d) \in \text{set}\ \mathit{buf}.\ t \leq \mathit{wm} \\
\qquad \texttt{then let}\ \mathit{out} = \text{filter}\ (\lambda (t, \_).\ t \leq \mathit{wm})\ \mathit{buf}; \\
\qquad\qquad\qquad \mathit{buf'} = \text{filter}\ (\lambda (t, \_).\ \neg\ t \leq \mathit{wm})\ \mathit{buf} \\
\qquad\quad \texttt{in}\ (\text{batch\_op}\ \mathit{buf'}, [\text{DT}\ \mathit{wm}\ \mathit{out}, \text{WM}\ \mathit{wm}]) \\
\qquad \texttt{else}\ (\text{batch\_op}\ \mathit{buf}, [\text{WM}\ \mathit{wm}]))
\end{array}
$$

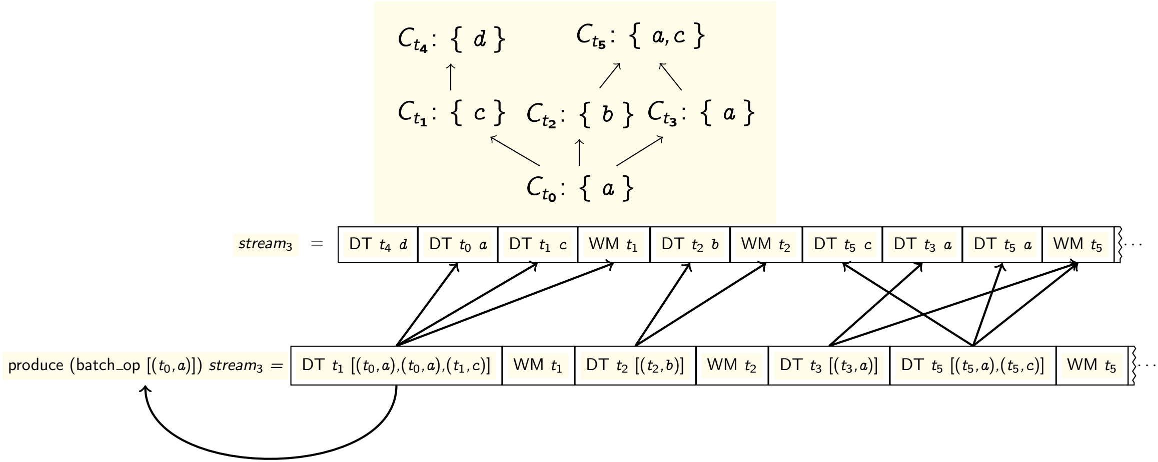
Building Blocks: Batch Operator

- Building Blocks: reusable operators
  - Batching and incremental computations
- batch_op : produces batches of accumulated data

$$
\begin{array}{l}
\textbf{corec}\ \mathsf{batch\_op}\ \textbf{where} \\
\quad \mathsf{batch\_op}\ \mathit{buf} = \mathsf{Logic}\ (\lambda \mathit{ev}.\ \mathtt{case}\ \mathit{ev}\ \mathtt{of}\ \mathsf{DT}\ t\ d \Rightarrow (\mathsf{batch\_op}\ (\mathit{buf}\ @\ [(t,\ d)]),\ []) \\
\quad \mid \mathsf{WM}\ \mathit{wm} \Rightarrow \mathtt{if}\ \exists (t,\ d) \in \mathsf{set}\ \mathit{buf}.\ t \le \mathit{wm} \\
\qquad \mathtt{then}\ \mathtt{let}\ \mathit{out} = \mathsf{filter}\ (\lambda(t,\ \_).\ t \le \mathit{wm})\ \mathit{buf}; \\
\qquad\qquad\qquad \mathit{buf'} = \mathsf{filter}\ (\lambda(t,\ \_).\ \neg\ t \le \mathit{wm})\ \mathit{buf} \\
\qquad\qquad \mathtt{in}\ (\mathsf{batch\_op}\ \mathit{buf'},\ [\mathsf{DT}\ \mathit{wm}\ \mathit{out},\ \mathsf{WM}\ \mathit{wm}]) \\
\qquad \mathtt{else}\ (\mathsf{batch\_op}\ \mathit{buf},\ [\mathsf{WM}\ \mathit{wm}]))
\end{array}
$$

Batch Operator: Soundness

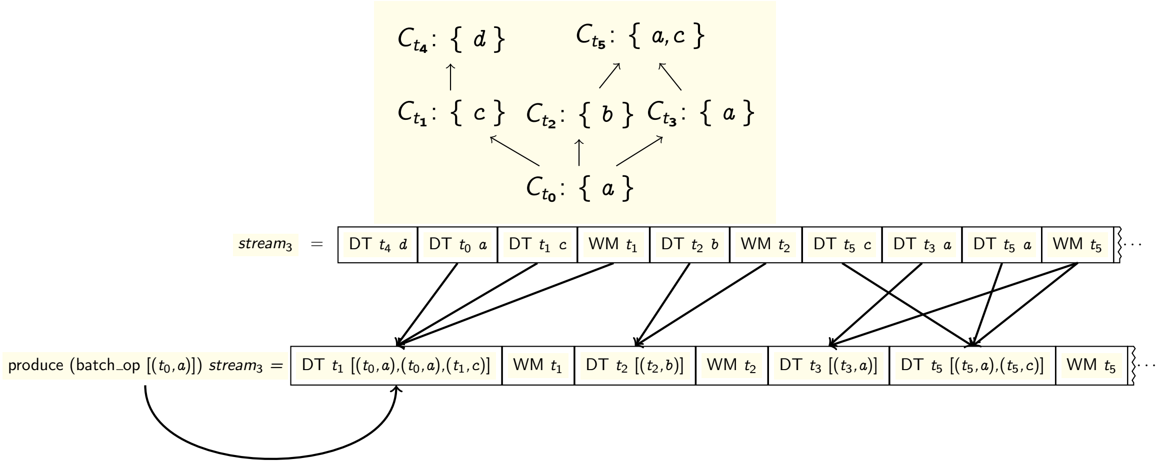
- Given a monotone time-aware stream



- Proof: lset induction, produce_1 induction, and generalization with skip_op

Batch Operator: Completeness

- Given a monotone and productive time-aware stream
- if not finite



- Proof: induction over the position (nat) of the element in the input, and soundness of batch_op

Batch Operator: Monotone and productive preservation

- The operators must preserve monotone and productive, so we can compose it with something that needs these properties!

$$\text{monotone } \mathit{lxs}\ W \longrightarrow \text{monotone } (\text{produce } (\text{batch\_op } \mathit{buf})\ \mathit{lxs})\ W \tag{1}$$

$$\text{productive } \mathit{lxs} \longrightarrow \text{productive } (\text{produce } (\text{batch\_op } \mathit{buf})\ \mathit{lxs}) \tag{2}$$

- Proof: coinduction up to congruence

Building Blocks: Incremental Operator

- Incremental computations
- incr_op : produces accumulated batches of accumulated data

$$
\begin{array}{l}
\textbf{corec}\ \mathsf{incr\_op}\ \textbf{where} \\
\quad \mathsf{incr\_op}\ \mathit{buf} = \mathsf{Logic}\ (\lambda\ \mathit{ev}.\ \mathtt{case}\ \mathit{ev}\ \mathtt{of}\ \mathsf{DT}\ \mathit{wm}\ \mathit{batch} \Rightarrow \\
\qquad\quad \mathtt{let}\ \mathit{out} = \mathsf{map}\ (\lambda t.\ \mathsf{DT}\ t\ (\mathit{buf}\ @\ \mathit{batch}))\ (\mathsf{remdups}\ (\mathsf{map}\ \mathsf{fst}\ \mathit{batch})) \\
\qquad\quad \mathtt{in}\ (\mathsf{incr\_op}\ (\mathit{buf}\ @\ \mathit{batch}), \mathit{out}) \\
\quad \mid \mathsf{WM}\ \mathit{wm} \Rightarrow\ (\mathsf{incr\_op}\ \mathit{buf}, [\mathsf{WM}\ \mathit{wm}]))
\end{array}
$$

Incremental Operator: Soundness

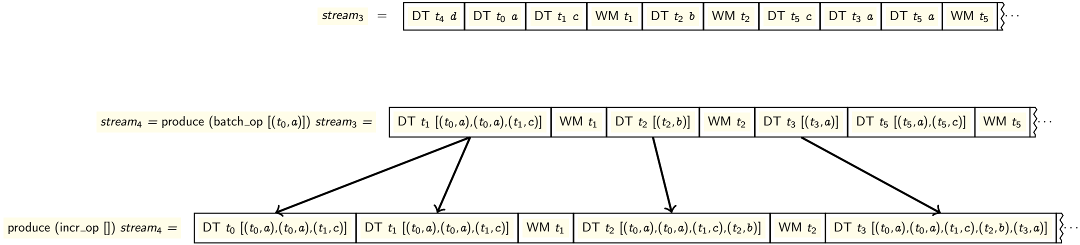
$stream_3$ = | DT t_4 d | DT t_0 a | DT t_1 c | WM t_1 | DT t_2 b | WM t_2 | DT t_5 c | DT t_3 a | DT t_5 a | WM t_5 | ⋯

$stream_4$ = produce (batch_op $[(t_0, a)]$) $stream_3$ = | DT t_1 $[(t_0,a),(t_0,a),(t_1,c)]$ | WM t_1 | DT t_2 $[(t_2,b)]$ | WM t_2 | DT t_3 $[(t_3,a)]$ | DT t_5 $[(t_5,a),(t_5,c)]$ | WM t_5 | ⋯

produce (incr_op $[]$) $stream_4$ = | 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_2 $[(t_0,a),(t_0,a),(t_1,c),(t_2,b)]$ | WM t_2 | DT t_3 $[(t_0,a),(t_0,a),(t_1,c),(t_2,b),(t_3,a)]$ | ⋯

- Proof: produce$_1$ induction, and generalization with skip_op

Incremental Operator: Completeness



- Proof: induction over the position (nat) of the element in the input

Incremental Operator: Monotone and productive preservation

$$\text{monotone } lxs\ W \longrightarrow \text{monotone (produce (incr\_op []) } lxs\text{) } W \tag{3}$$

$$\text{productive } lxs \longrightarrow \text{productive (produce (incr\_op []) } lxs\text{)} \tag{4}$$

- Proof: coinduction up to congruence

Compositional Reasoning

- batch_op and incr_op can be composed

  definition incr_batch_op *buf1 buf2* = comp_op (batch_op *buf1*) (incr_op *buf2*)

- Soundness, completeness, and monotone and productive preservation

Case Study

Histogram

- A histogram count the elements of a collection
- Incremental histogram: timestamps smaller or equal
- $H_{t_5} = C_{t_0} + C_{t_0} + C_{t_1} + C_{t_2} + C_{t_5}$
- paths to t_5: $\{t_0, t_2\}$ and $\{t_0, t_3\}$

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

Histogram Operator

corec map_op **where** map_op f = Logic (λ $ev.$ case ev of
WM $wm \Rightarrow$ (map_op f, [WM wm]) | DT t $d \Rightarrow$ (map_op f, [DT t (f t d)]))

abbreviation data_at_from_list xs $t \equiv$ map snd (filter (λ (t', d) . $t' = t$) xs)
definition coll xs t = mset (data_at_from_list xs t)

definition incr_coll t xs = coll xs t + mset (concat (map
(λ x . concat (map (λ t' . coll xs t') x)) (paths (remdups (map fst xs)) t)))

definition incr_hist_op $buf1$ $buf2$ =
comp_op (incr_batch_op $buf1$ $buf2$) (map_op incr_coll)

Histogram Operator: Soundness, Completeness, Monotone and Productive Preservation

- Given a monotone and productive time-aware stream

$$stream_3 = \boxed{\text{DT } t_4\ d} \boxed{\text{DT } t_0\ a} \boxed{\text{DT } t_1\ c} \boxed{\text{WM } t_1} \boxed{\text{DT } t_2\ b} \boxed{\text{WM } t_2} \boxed{\text{DT } t_5\ c} \boxed{\text{DT } t_3\ a} \boxed{\text{DT } t_5\ a} \boxed{\text{WM } t_5} \cdots$$

$$\text{produce (incr\_hist\_op [] [])}\ stream_3 = \boxed{\text{DT } t_0\ (\text{mset } [a])} \boxed{\text{DT } t_1\ (\text{mset } [a, c])} \boxed{\text{WM } t_1} \boxed{\text{DT } t_2\ (\text{mset } [a, b])} \boxed{\text{WM } t_2} \boxed{\text{DT } t_3\ (\text{mset } [a, a])} \boxed{\text{DT } t_5\ (\text{mset } [a, a, a, b, c])} \boxed{\text{WM } t_5} \cdots$$

- Proof: soundness, completeness, monotone and productive preservation of incr_batch_op

Efficient Histogram Operator

- We show a equivalent efficient histogram operator for timestamp in total order
- Equivalent only for monotone time-aware stream

$$
\begin{array}{l}
\textbf{corec}\ \text{incr\_hist\_op}'\ \textbf{where} \\
\quad \text{incr\_hist\_op}'\ H\ \mathit{buf} = \text{Logic}\ (\lambda\ \mathit{ev}.\ \texttt{case}\ \mathit{ev}\ \texttt{of} \\
\quad\quad \text{DT}\ (t{::}\_{::}\mathit{linorder})\ d \Rightarrow (\text{incr\_hist\_op}'\ H\ (\mathit{buf}\ @\ [(t,\ d)]),\ []) \\
\quad \mid \text{WM}\ \mathit{wm} \Rightarrow \texttt{if}\ \exists\ (t,\ d) \in \text{set}\ \mathit{buf}\ .\ t \leq \mathit{wm} \\
\quad\quad \texttt{then}\ \texttt{let}\ \mathit{out} = \text{filter}\ (\lambda\ (t,\ \_).\ t \leq \mathit{wm})\ \mathit{buf}\ \texttt{in} \\
\quad\quad\quad \texttt{let}\ \mathit{buf}' = \text{filter}\ (\lambda\ (t,\ \_).\ t > \mathit{wm})\ \mathit{buf}\ \texttt{in} \\
\quad\quad\quad \texttt{let}\ \text{ts} = \text{remdups}\ ((\text{map fst}\ \mathit{out}))\ \texttt{in} \\
\quad\quad\quad \texttt{let}\ \mathit{Hs} = \text{map} \\
\quad\quad\quad\quad (\lambda t.\ \text{DT}\ t\ (H + (\text{mset}\ (\text{map snd}\ (\text{filter}\ (\lambda(t',\ \_).\ t' \leq t)\ \mathit{out})))))) \\
\quad\quad\quad\quad \text{ts}\ \texttt{in} \\
\quad\quad\quad (\text{incr\_hist\_op}'\ (H + (\text{mset}\ (\text{map snd}\ \mathit{out})))\ \mathit{buf}',\ \mathit{Hs}\ @\ [\text{WM}\ \mathit{wm}]) \\
\quad\quad \texttt{else}\ (\text{incr\_hist\_op}'\ H\ \mathit{buf},\ []))
\end{array}
$$

Join

- Use the `sum` type to represent two stream as one
- Partial order for the `sum`: left compares with left, right compares with right
- Defined using `incr_batch_op`
- Soundness, Completeness, Monotone

Next Steps

Next Steps

- Feedback loop
- Exit argument
- connect to the Isabelle-LLVM refinement framework

Questions, comments and suggestions