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
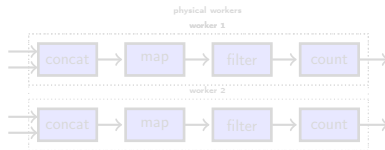
What is this PhD/Status seminar about?
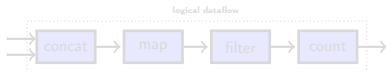
- Distributed Systems
  - Stream processing frameworks
    - Dataflow models
    - Time-Aware Computations
- Formal Methods
  - Verification using proof assistants
    - Isabelle proofs
    - Verified + executable + efficient code
- Formalization of Time-Aware Stream Processing

Introduction

- 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
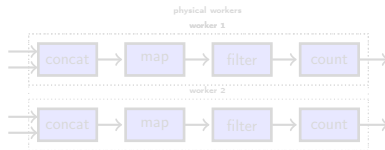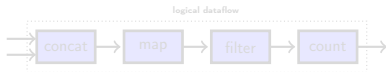

- Highly Parallel

- 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
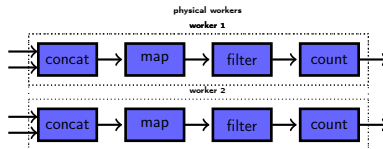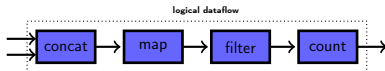

- Highly Parallel

- 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

- 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 ($\dots$)))

- 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

- Datatypes and Codatatypes

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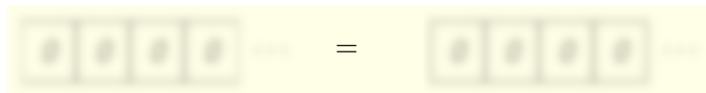 **for** map: lmap **where** ltl LNil = LNil

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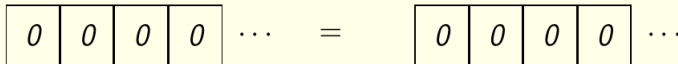 - LCons *1* (LCons *2* (LCons *3* LNil))
  - LCons *0* (LCons *0* (LCons *0* (...)))
- Induction principle assuming membership in the lazy list
- Coinductive principle for lazy list equality:
  - Show that there is a pair of goggles that makes them to look the same, which implies that:
    - The first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

- Recursion

  fun lshift $:: \, 'a \, list \Rightarrow 'a \, llist \Rightarrow 'a \, llist$ (*infixr* @@ *65*) **where**
    lshift [] lxs = lxs
  | lshift $(x \mathbin{\#} xs)$ lxs = LCons x (lshift xs lxs)

- While Combinator

  definition while_option $:: \, ('a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a \, option$ **where**
  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

- 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: 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

> **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 ($\ldots$))))

- 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 ($\ldots$)))) (LCons 1 (LCons (2 ($\ldots$))))

- 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 :: $'a \Rightarrow\ 'a$ 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

Lazy Lists Processors

- Operator as a codatatype
  - Taking $'i$ as the input type, and $'o$ as the output type:
    codatatype $('o, 'i)$ $op =$ Logic $(\text{apply}: ('i \Rightarrow ('o, 'i)$ $op \times 'o$ $list))$
  - 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:
    codatatype $('o, 'i)\ op =$ Logic (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

  definition produce$_1$ *op lxs* = while_option . . .

  corec produce **where**
   produce *op lxs* = (case produce$_1$ *op lxs* of None $\Rightarrow$ LNil
    | Some (*op'*, *x*, *xs*, *lxs'*) $\Rightarrow$ LCons *x* (*xs* @@ produce *op' lxs'*))

- produce$_1$ has an induction principle based on the while invariant rule

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

  definition $produce_1$ *op lxs* = while_option . . .
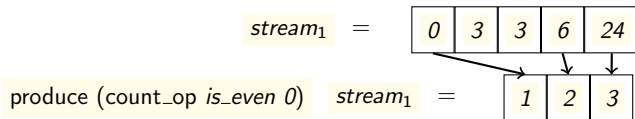
  corec produce **where**
    produce *op lxs* = (case $produce_1$ *op lxs* of None $\Rightarrow$ LNil
      | Some (op', x, xs, lxs') $\Rightarrow$ LCons x (xs @@ produce op' lxs'))

- $produce_1$ has an induction principle based on the while invariant rule

- Example:

corec count_op **where** count_op *P n* =
  Logic ($\lambda e.$ **if** *P e* **then** (count_op *P* (*n* + *1*), [*n+1*]) **else** (count_op *P n*, []))

$stream_1$ =

0	3	3	6	24

produce (count_op *is_even 0*) $stream_1$ =

1	2	3

Sequential Composition

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

definition fproduce op xs = fold (λe (op, out).
  let (op', out') = apply op e in (op', out @ out')) xs (op, [])
corec comp_op **where**
  comp_op op_1 op_2 = Logic (λev.
    let (op_1', out) = apply op_1 ev; (op_2', out') = fproduce op_2 out
    in (comp_op op_1' op_2', out'))

Sequential Composition: Correctness

- Correctness:

  produce (comp_op op_1 op_2) lxs = produce op_2 (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 ($\lambda ev.$ let (op', out) = apply op ev in
        if length $out < n$ then (skip_op op' $(n -$ length $out)$, [])
        else $(op',$ drop n $out))$

    item Correctness

      produce (skip_op op n) lxs = ldropn produce op lxs

  - Proof: Coinduction up to congruence for lazy list equality

Sequential Composition: Correctness

- Correctness:

  produce (comp_op op_1 op_2) lxs = produce op_2 (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 ($\lambda ev.$ let (op', out) = apply op ev in
        if length out < n then (skip_op op' (n − length out), [])
        else (op', drop n out))

    item Correctness

      produce (skip_op op n) lxs = ldropn produce op lxs

  - Proof: Coinduction up to congruence for lazy list equality

- Correctness:

  produce (comp_op op_1 op_2) lxs = produce op_2 (produce op_1 lxs)

- Proof: coinduction principle for lazy list equality and produce₁ 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))

    item Correctness

    produce (skip_op op n) lxs = ldropn produce op lxs

  - Proof: Coinduction up to congruence for lazy list equality

Sequential Composition: Correctness

- Correctness:

  produce (comp_op op_1 op_2) lxs = produce op_2 (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 ($\lambda ev.$ let (op', out) = apply op ev in
        if length $out < n$ then (skip_op op' ($n -$ length out), [])
        else (op', drop n out))

    item Correctness

    produce (skip_op op n) lxs = ldropn produce op lxs

  - Proof: Coinduction up to congruence for lazy list equality

Time-Aware Operators

- Time-Aware lazy lists

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

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

Time-Aware Streams

- Time-Aware lazy lists

  datatype (′t::order, ′d) event = DT (tmp: ′t) (data: ′d) | WM (wmk: ′t)

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

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**
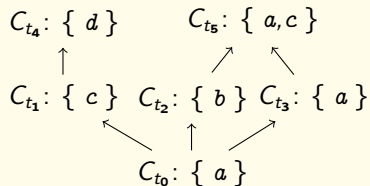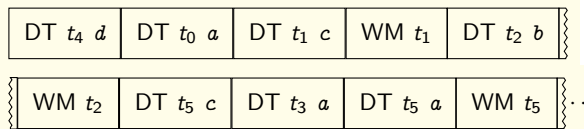>   LNil: monotone LNil W
> | LConsR: $(\forall wm' \in W. \neg wm' \geq wm) \longrightarrow$ monotone lxs $(\{wm\} \cup W) \longrightarrow$
>   monotone (LCons (WM wm) lxs) W
> | LConsL: $(\forall wm \in W. \neg wm \geq t) \longrightarrow$ monotone lxs $W \longrightarrow$
>   monotone (LCons (DT t d) lxs) W

- Up to congruence coinduction principle
- Example:

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: ¬ lfinite lxs $\longrightarrow$ ($\exists u \in$ vimage WM (lset lxs). $u \geq t$) $\longrightarrow$
      productive lxs $\longrightarrow$ productive (LCons (DT t d) lxs)
    | SkipWM: ¬ lfinite lxs $\longrightarrow$ productive lxs $\longrightarrow$
      productive (LCons (WM t) lxs)

  - Up to congruence coinduction principle

- 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 ⟶ productive lxs
    | EnvWM: ¬ lfinite lxs ⟶ (∃u ∈ vimage WM (lset lxs). u ≥ t) ⟶
      productive lxs ⟶ productive (LCons (DT t d) lxs)
    | SkipWM: ¬ lfinite lxs ⟶ productive lxs ⟶
      productive (LCons (WM t) lxs)

  - Up to congruence coinduction principle

- 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*: ¬ lfinite $lxs \longrightarrow (\exists u \in$ vimage WM (lset lxs). $u \geq t) \longrightarrow$
     productive $lxs \longrightarrow$ productive (LCons (DT t d) lxs)
   | *SkipWM:* ¬ lfinite $lxs \longrightarrow$ productive $lxs \longrightarrow$
     productive (LCons (WM t) lxs)

  - Up to congruence coinduction principle

- 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: ¬ lfinite lxs $\longrightarrow$ ($\exists u \in$ vimage WM (lset lxs). $u \geq t$) $\longrightarrow$
  >   productive lxs $\longrightarrow$ productive (LCons (DT t d) lxs)
  > | SkipWM: ¬ 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

```
corec batch_op where
 batch_op buf = Logic (λev. case ev of DT t d ⇒ (batch_op (buf @ [(t, d)]), [])
 | WM wm ⇒ if ∃(t, d) ∈ set buf. t ≤ wm
    then let out = filter (λ(t, _). t ≤ wm) buf;
            buf' = filter (λ(t, _). ¬ t ≤ wm) buf
        in (batch_op buf', [DT wm out, WM wm])
    else (batch_op buf, [WM wm]))
```

Building Blocks: Batch Operator

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

```
corec batch_op where
  batch_op buf = Logic (λev. case ev of DT t d ⇒ (batch_op (buf @ [(t, d)]), [])
  | WM wm ⇒ if ∃(t, d) ∈ set buf. t ≤ wm
      then let out = filter (λ(t, _). t ≤ wm) buf;
             buf' = filter (λ(t, _). ¬ t ≤ wm) buf
        in (batch_op buf', [DT wm out, WM wm])
      else (batch_op buf, [WM wm]))
```
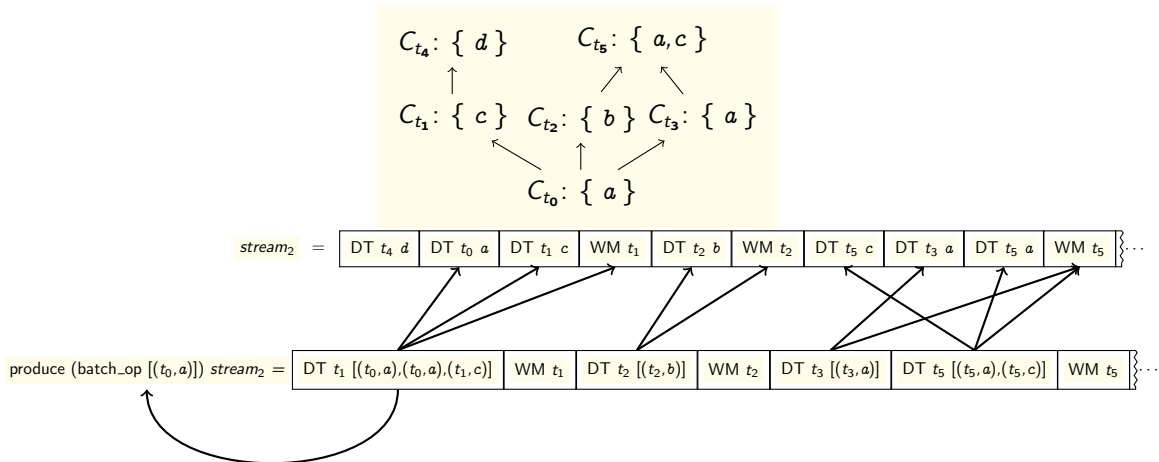
Batch Operator: Soundness

- Given a monotone time-aware stream

$$C_{t_4}: \{\, d \,\} \qquad C_{t_5}: \{\, a, c \,\}$$

$$C_{t_1}: \{\, c \,\} \quad C_{t_2}: \{\, b \,\} \quad C_{t_3}: \{\, a \,\}$$

$$C_{t_0}: \{\, a \,\}$$

$stream_2 \ = \ $ | 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 | $\cdots$

produce (batch_op [(t_0, a)]) $stream_2 \ = \ $ | 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 | $\cdots$

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

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

$C_{t_4}: \{ d \}$ $C_{t_5}: \{ a,c \}$

$C_{t_1}: \{ c \}$ $C_{t_2}: \{ b \}$ $C_{t_3}: \{ a \}$

$C_{t_0}: \{ a \}$

$stream_2 =$ | 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 | $\cdots$

produce (batch_op [(t_0,a)]) $stream_2 =$ | 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 | $\cdots$
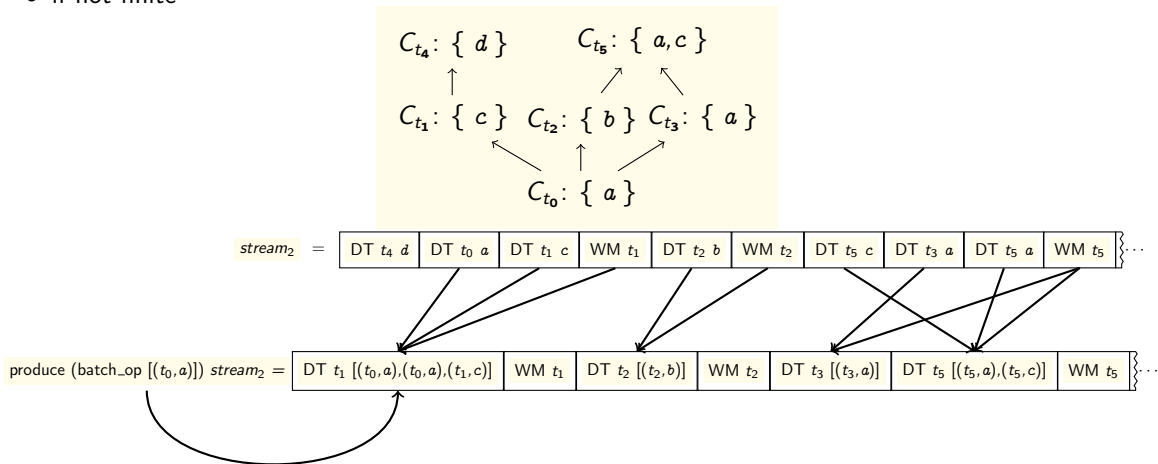
- Proof: induction over the position (nat) of the element in the input, and soundness of batch_op

Batch Operator: Monotone and productive preservation

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

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

$$\text{productive } lxs \longrightarrow \text{productive (produce (batch\_op } buf) \; lxs) \qquad (2)$$

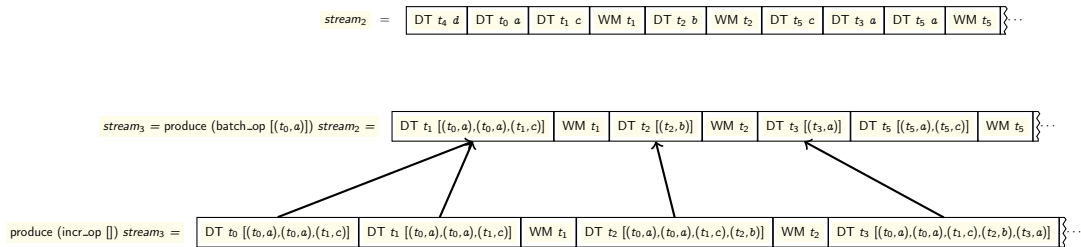- Proof: coinduction up to congruence

Building Blocks: Incremental Operator

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

  corec incr_op **where**
   incr_op *buf* = Logic (λ *ev*. case *ev* of DT *wm batch* $\Rightarrow$
      let *out* = map (λt. DT *t* (*buf* @ *batch*)) (remdups (map fst *batch*))
      in (incr_op (*buf* @ *batch*), *out*)
   | WM *wm* $\Rightarrow$ (incr_op *buf*, [WM *wm*]))

- Proof: produce₁ induction, and generalization with skip_op

Incremental Operator: Completeness

$stream_2 =$ | 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 $\cdots$

$stream_3 = $ produce (batch_op $[(t_0,a)]$) $stream_2 =$ | 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 $\cdots$

produce (incr_op $[]$) $stream_3 =$ | 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 $[(t_0,a),(t_0,a),(t_1,c),(t_2,b),(t_3,a)]$ $\cdots$

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

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

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

- 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 $\{t_0, t_3\}$

C_{t_4}: $\{\ d\ \}$ C_{t_5}: $\{\ a, c\ \}$ H_{t_4}: $\{\ a, c, d\ \}$ H_{t_5}: $\{\ a, a, a, a, b, c\ \}$

C_{t_1}: $\{\ c\ \}$ C_{t_2}: $\{\ b\ \}$ C_{t_3}: $\{\ a\ \}$ H_{t_1}: $\{\ c, a\ \}$ H_{t_2}: $\{\ b, a\ \}$ H_{t_3}: $\{\ a, a\ \}$

C_{t_0}: $\{\ a\ \}$ H_{t_0}: $\{\ a\ \}$
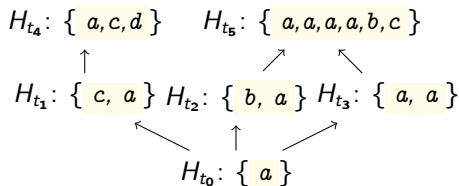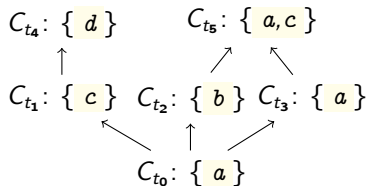
Histogram Operator

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)]))

definition incr_coll t xs = mset . . .

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

- Given a monotone and productive time-aware stream

$stream_2$ = | 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 | $\cdots$

produce (incr_hist_op [] []) $stream_2$ = | DT t_0 (mset [a]) | DT t_1 (mset [a,c]) | WM t_1 | DT t_2 (mset [a,b]) | WM t_2 | DT t_3 (mset [a,a]) | DT t_5 (mset [a,a,a,b,c]) | WM t_5 | $\cdots$

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

Efficient Histogram Operator

- Efficient histogram operator `incr_batch_op'` for timestamp in total order
  - State of the operator: last computed histogram, and buffer of newly accumulated data
- Equivalent `incr_batch_op` only for monotone time-aware stream (equivalence relation)

aa

s

n

v

c

l

r

s

m

x

Efficient Histogram Operator

- Efficient histogram operator `incr_batch_op'` for timestamp in total order
  - State of the operator: last computed histogram, and buffer of newly accumulated data
- Equivalent `incr_batch_op` only for monotone time-aware stream (equivalence relation)

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
  - WIP: Productive

Next Steps

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

Questions, comments and suggestions