

# Verified Time-Aware Stream Processing

Rafael Castro G. Silva

`razi@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

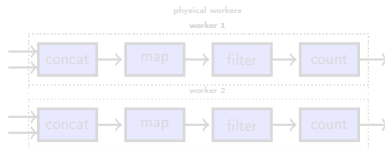
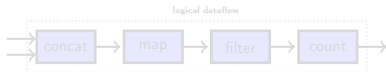
- Introduction
- Preliminaries
- Lazy Lists Processors
- Time-Aware Operators
- Case Study
- Next Steps

# Introduction

# Stream Processing

- Stream Processing: Abstraction for processing data when the input is not completely presented in the beginning 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



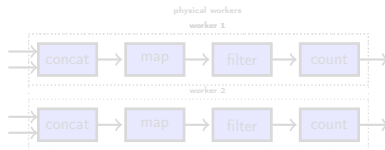
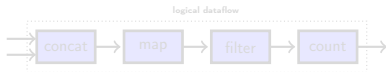
- Time-Aware Computations
  - Timestamps: Metadata associating the data with some data collection
  - Watermarks: Metadata indicating the completion of a data collection

# Stream Processing

- Stream Processing: Abstraction for processing data when the input is not completely presented in the beginning 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 beginning 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



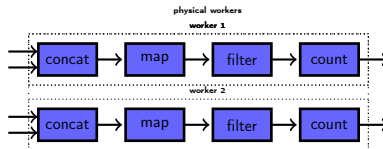
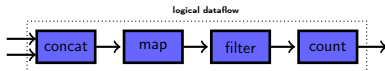
Cloud  
DataFlow



Flink



- Highly Parallel



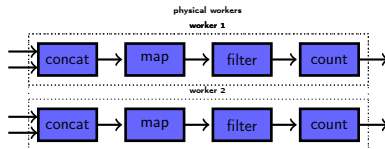
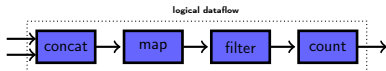
- Time-Aware Computations
  - Timestamps: Metadata associating the data with some data collection
  - Watermarks: Metadata indicating the completion of a data collection

# Stream Processing

- Stream Processing: Abstraction for processing data when the input is not completely presented in the beginning 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

- 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 = Inul: LNil | LCons (lhd: 'a) (ltl: 'a llist)  
for map: lmap where ltl LNil = LNil
```

- Examples:

- LNil

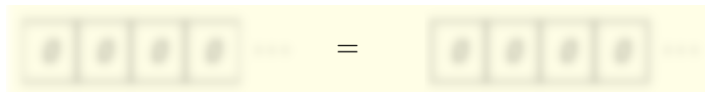
- LCons 1 (LCons 2 (LCons 3 LNil))

- 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

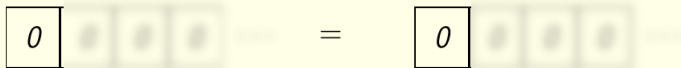
- LCons 1 (LCons 2 (LCons 3 LNil))

- 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


$$\boxed{0} \dots = \boxed{0} \dots$$

# 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

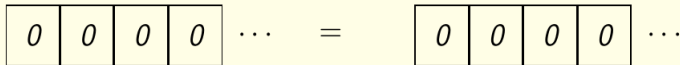
- LCons 1 (LCons 2 (LCons 3 LNil))

- 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



- Recursion

```
fun lshift :: 'a list  $\Rightarrow$  'a llist  $\Rightarrow$  'a llist (infixr @@ 65) where  
  lshift [] lxs = lxs  
| lshift (x # 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 = ...
```

- 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:

```
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 xs lys = case xs of LNil  $\Rightarrow$  lys | LCons x xs'  $\Rightarrow$  LCons x (lapp xs' lys)
```

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

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

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

- 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 xs lys = case xs of LNil  $\Rightarrow$  lys | LCons x xs'  $\Rightarrow$  LCons x (lapp xs' lys)
```

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

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

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

- 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 xs lys = case xs of LNil  $\Rightarrow$  lys | LCons x xs'  $\Rightarrow$  LCons x (lapp xs' lys)
```

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

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

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

- 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 lxs  
  | LCons_lprefix: lprefix lxs lxs  $\Rightarrow$  lprefix (LCons x lxs) (LCons x lxs)  
  
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 (...))))
```

- Coinductive predicate
  - Infinite number of introduction rule applications

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

- Coinductive predicate
  - Infinite number of introduction rule applications

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

# 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

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

**definition**  $\text{produce}_1' \text{ op } lxs = \text{while\_option}$

$(\lambda(\text{op}, lxs). \neg \text{Inull } lxs \wedge \text{snd } (\text{apply } \text{op } (\text{lhs } lxs))) = [])$

$(\lambda(\text{op}, lxs). (\text{fst } (\text{apply } \text{op } (\text{lhs } lxs)), \text{tl } lxs)) (\text{op}, lxs)$

**definition**  $\text{produce}_1 \text{ op } lxs =$

$(\text{case } \text{produce}_1' \text{ op } lxs \text{ of } \text{None} \Rightarrow \text{None}$

$| \text{Some } (\text{op}', lxs') \Rightarrow \text{if } \text{Inull } lxs' \text{ then } \text{None} \text{ else}$

$\text{let } (\text{op}'', \text{out}) = \text{apply } \text{op}' (\text{lhs } lxs') \text{ in } \text{Some } (\text{op}'', \text{hd } \text{out}, \text{tl } \text{out}, \text{tl } lxs'))$

**corec** **produce** **where**

$\text{produce } \text{op } lxs = (\text{case } \text{produce}_1 \text{ op } lxs \text{ of } \text{None} \Rightarrow \text{LNil}$

$| \text{Some } (\text{op}', x, xs, lxs') \Rightarrow \text{LCons } x (xs \text{ @@ } \text{produce } \text{op}' lxs'))$

- $\text{produce}_1$  has an induction principle based on the while invariant rule

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

**definition**  $\text{produce}_1' \text{ op } lxs = \text{while\_option}$

$(\lambda(\text{op}, lxs). \neg \text{Inull } lxs \wedge \text{snd } (\text{apply } \text{op } (\text{lhs } lxs))) = [])$

$(\lambda(\text{op}, lxs). (\text{fst } (\text{apply } \text{op } (\text{lhs } lxs)), \text{tl } lxs)) (\text{op}, lxs)$

**definition**  $\text{produce}_1 \text{ op } lxs =$

$(\text{case } \text{produce}_1' \text{ op } lxs \text{ of } \text{None} \Rightarrow \text{None}$

$| \text{Some } (\text{op}', lxs') \Rightarrow \text{if } \text{Inull } lxs' \text{ then } \text{None} \text{ else}$

$\text{let } (\text{op}'', \text{out}) = \text{apply } \text{op}' (\text{lhs } lxs') \text{ in } \text{Some } (\text{op}'', \text{hd } \text{out}, \text{tl } \text{out}, \text{tl } lxs'))$

**corec** **produce** **where**

$\text{produce } \text{op } lxs = (\text{case } \text{produce}_1 \text{ op } lxs \text{ of } \text{None} \Rightarrow \text{LNil}$

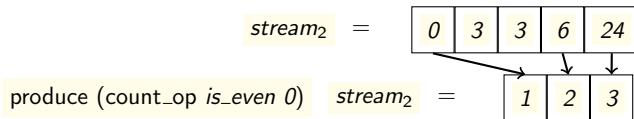
$| \text{Some } (\text{op}', x, xs, lxs') \Rightarrow \text{LCons } x (xs \text{ @@ } \text{produce } \text{op}' lxs'))$

- $\text{produce}_1$  has an induction principle based on the while invariant rule

# Operators: Count

- 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, []))
```





# Sequential Composition

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

```
definition fproduce  $op\ xs = \text{fold } (\lambda e\ (op,\ out)).$   
  let  $(op',\ out') = \text{apply } op\ e\ \text{in } (op',\ out\ @\ out')$  xs (op, [])  
corec comp_op where  
  comp_op  $op_1\ op_2 = \text{Logic } (\lambda ev.$   
    let  $(op_1',\ out) = \text{apply } op_1\ ev;$   $(op_2',\ out') = \text{fproduce } op_2\ out$   
    in  $(\text{comp\_op } op_1'\ op_2',\ out')$ 
```

# Sequential Composition: Correctness

- Correctness:

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

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

$\text{corec skip\_op}$  where

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

- Correctness: Coinduction up to congruence for lazy list equality

# Sequential Composition: Correctness

- Correctness:

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

- Proof: coinduction principle for lazy list equality and  $\text{produce}_1$  induction principle

- Generalization: we must be able to reason about elements in arbitrary positions

$\text{corec skip\_op}$  where

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

- Correctness: Coinduction up to congruence for lazy list equality

# Sequential Composition: Correctness

- Correctness:

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

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

**corec skip\_op where**

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

- Correctness: Coinduction up to congruence for lazy list equality

# Sequential Composition: Correctness

- Correctness:

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

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

**corec skip\_op where**

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

- Correctness: 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  $:: ('t::\text{order}, 'd) \text{ event llist} \Rightarrow 't \text{ set} \Rightarrow \text{bool}$  where

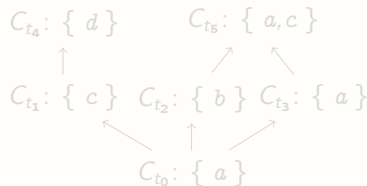
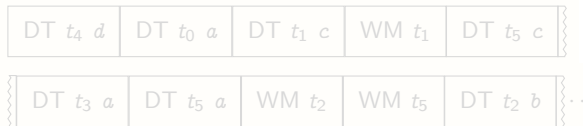
LNil: monotone LNil  $W$

| LConsR:  $(\forall wm' \in W. \neg wm' \geq wm) \longrightarrow \text{monotone } lxs (\{wm\} \cup W) \longrightarrow$   
 monotone (LCons (WM  $wm$ )  $lxs$ )  $W$

| LConsL:  $(\forall wm \in W. \neg wm \geq t) \longrightarrow \text{monotone } lxs W \longrightarrow$   
 monotone (LCons (DT  $t d$ )  $lxs$ )  $W$

- Up to congruence coinduction principle
- Example:

$stream_3 =$



# Monotone Time-Aware Streams

- Monotone: watermarks do not go back in time

**coinductive** monotone  $:: ('t::order, 'd) \text{ event llist} \Rightarrow 't \text{ set} \Rightarrow \text{bool}$  where

LNil: monotone LNil  $W$

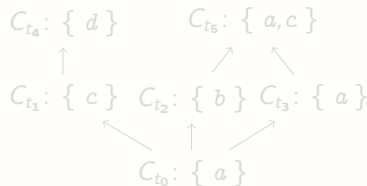
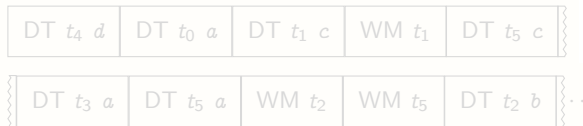
| LConsR:  $(\forall wm' \in W. \neg wm' \geq wm) \longrightarrow \text{monotone } lxs (\{wm\} \cup W) \longrightarrow$   
 monotone (LCons (WM  $wm$ )  $lxs$ )  $W$

| LConsL:  $(\forall wm \in W. \neg wm \geq t) \longrightarrow \text{monotone } lxs W \longrightarrow$   
 monotone (LCons (DT  $t d$ )  $lxs$ )  $W$

- Up to congruence coinduction principle

- Example:

$stream_3 =$



# Monotone Time-Aware Streams

- Monotone: watermarks do not go back in time

**coinductive** monotone  $:: ('t::\text{order}, 'd) \text{ event llist} \Rightarrow 't \text{ set} \Rightarrow \text{bool}$  where

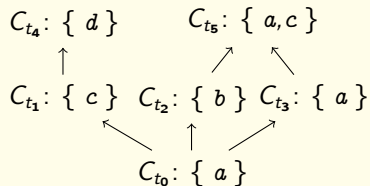
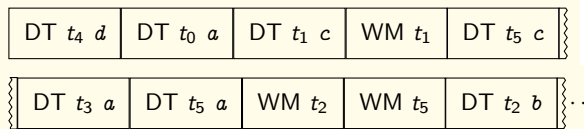
LNil: monotone LNil  $W$

| LConsR:  $(\forall wm' \in W. \neg wm' \geq wm) \longrightarrow \text{monotone } lxs (\{wm\} \cup W) \longrightarrow$   
 monotone (LCons (WM  $wm$ )  $lxs$ )  $W$

| LConsL:  $(\forall wm \in W. \neg wm \geq t) \longrightarrow \text{monotone } lxs W \longrightarrow$   
 monotone (LCons (DT  $t d$ )  $lxs$ )  $W$

- Up to congruence coinduction principle
- Example:

$stream_3 =$



# 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  $lxs \rightarrow$  productive  $lxs$

| EnvWM:  $\neg$  lfinite  $lxs \rightarrow (\exists u \in \text{vimage WM } (\text{lset } lxs). u \geq t) \rightarrow$   
productive  $lxs \rightarrow$  productive (LCons (DT  $t$   $d$ )  $lxs$ )

| SkipWM:  $\neg$  lfinite  $lxs \rightarrow$  productive  $lxs \rightarrow$   
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  $lxs \rightarrow$  productive  $lxs$

| EnvWM:  $\neg$  lfinite  $lxs \rightarrow (\exists u \in \text{vimage WM } (\text{lset } lxs). u \geq t) \rightarrow$   
productive  $lxs \rightarrow$  productive (LCons (DT  $t$   $d$ )  $lxs$ )

| SkipWM:  $\neg$  lfinite  $lxs \rightarrow$  productive  $lxs \rightarrow$   
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

$\text{LFinite}: \text{lfinite } lxs \longrightarrow \text{productive } lxs$

|  $\text{EnvWM}: \neg \text{lfinite } lxs \longrightarrow (\exists u \in \text{vimage WM } (\text{lset } lxs). u \geq t) \longrightarrow$   
 $\text{productive } lxs \longrightarrow \text{productive } (\text{LCons } (\text{DT } t \ d) \ lxs)$

|  $\text{SkipWM}: \neg \text{lfinite } lxs \longrightarrow \text{productive } lxs \longrightarrow$   
 $\text{productive } (\text{LCons } (\text{WM } t) \ lxs)$

- 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**

$\text{LFinite}: \text{lfinite } lxs \longrightarrow \text{productive } lxs$

|  $\text{EnvWM}: \neg \text{lfinite } lxs \longrightarrow (\exists u \in \text{vimage WM } (\text{lset } lxs). u \geq t) \longrightarrow$   
 $\text{productive } lxs \longrightarrow \text{productive } (\text{LCons } (\text{DT } t \ d) \ lxs)$

|  $\text{SkipWM}: \neg \text{lfinite } lxs \longrightarrow \text{productive } lxs \longrightarrow$   
 $\text{productive } (\text{LCons } (\text{WM } t) \ lxs)$

- 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

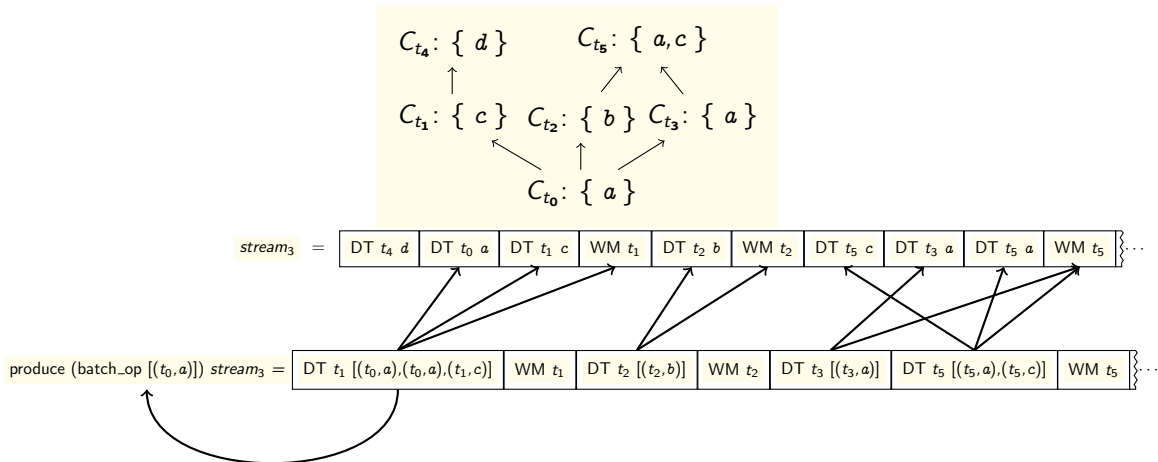
- Building Blocks: reusable operators
  - Batching and incremental computations
- `batch_op` : produces batches of accumulated data

**corec** `batch_op` **where**

```
batch_op buf = Logic ( $\lambda$ ev. case ev of DT  $t\ d \Rightarrow$  (batch_op (buf @ [(t, d)]), [])  
| WM wm  $\Rightarrow$  if  $\exists(t, d) \in \text{set } \text{buf}. t \leq wm$   
  then let out = filter ( $\lambda(t, \_). t \leq wm$ ) buf;  
    buf' = filter ( $\lambda(t, \_). \neg t \leq wm$ ) buf  
    in (batch_op buf', [DT wm out, WM wm])  
  else (batch_op buf, [WM wm]))
```

# Batch Operator: Soundness

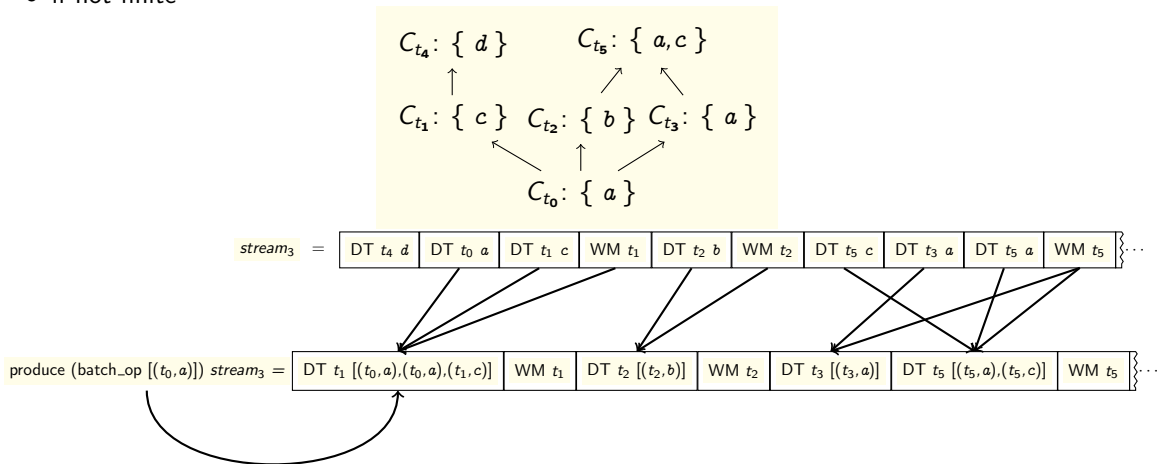
- Given a monotone time-aware stream



- Proof: `Iset` 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!

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

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

# Building Blocks: Incremental Operator

# Batch Operator: Soundness

# Batch Operator: Completeness

# Batch Operator: Monotone



# Batch Operator: Productive



## Case Study

# Histogram

# Histogram: Soundness

# Histogram: Completeness

# Histogram: Monotone

# Histogram: Productive



# Efficient Histogram

- Foo





# Join: Completeness

# Join: Monotone

## Next Steps

# Next Steps

Questions, comments and suggestions