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

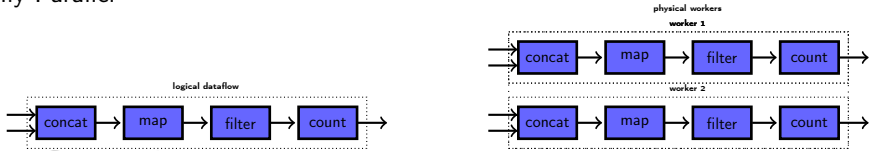
  - Highly Parallel

- 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

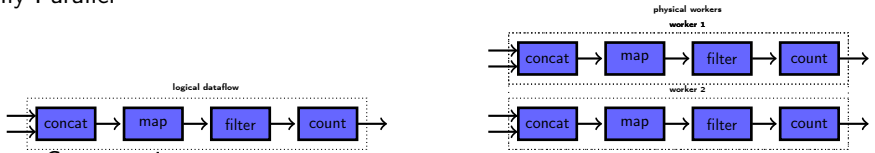


- Time-Aware Computations
  - Timestamps: Metadata associating the data with some data collection
  - Watermarks: Metadata indicating the completion of a data collection

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

  - 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
```

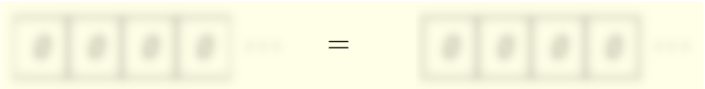
- Examples:
  - LNil
  - LCons *1* (LCons *2* (LCons *3* LNil))
  - LCons *0* (LCons *0* (LCons *0* (...)))
- Induction principle assuming membership in the lazy list
- Coinductive principle for lazy list equality:
  - Show that there is a pair of goggles that makes them to look the same, which implies 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

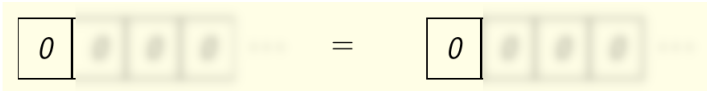


Isabelle/HOL: (Co)datatypes

- 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

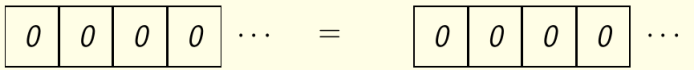


Isabelle/HOL: (Co)datatypes

- 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



Isabelle/HOL: Recursion and While Combinator

- Recursion

```
fun lshift :: 'a list ⇒ 'a llist ⇒ 'a llist (infixr @@ 65) where
  lshift [] lxs = lxs
| lshift (x # xs) lxs = LCons x (lshift xs lxs)
```

- While Combinator

```
definition while_option :: ('a ⇒ bool) ⇒ ('a ⇒ 'a) ⇒ 'a ⇒ '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 ⇒ 'a llist ⇒ 'a llist where
  lapp lxs lys = case lxs of LNil ⇒ lys | LCons x lxs' ⇒ 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
    [] ⇒ (case lxs of LNil ⇒ LNil | LCons x lxs' ⇒ LCons x lxs')
  | x#xs' ⇒ LCons x (xs' @@ lxs))
  by (auto split: list.splits llist.splits) (transfer_prover)

lconcat lxs = case lxs of LNil ⇒ LNil | LCons xs lxs' ⇒ 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 ⇒ 'a llist ⇒ 'a llist where
  lapp lxs lys = case lxs of LNil ⇒ lys | LCons x lxs' ⇒ 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
    [] ⇒ (case lxs of LNil ⇒ LNil | LCons x lxs' ⇒ LCons x lxs')
  | x#xs' ⇒ LCons x (xs' @@ lxs))
  by (auto split: list.splits llist.splits) (transfer_prover)

lconcat lxs = case lxs of LNil ⇒ LNil | LCons xs lxs' ⇒ 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 ⇒ 'a llist ⇒ 'a llist where
  lapp lxs lys = case lxs of LNil ⇒ lys | LCons x lxs' ⇒ 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
    [] ⇒ (case lxs of LNil ⇒ LNil | LCons x lxs' ⇒ LCons x lxs')
  | x#xs' ⇒ LCons x (xs' @@ lxs))
  by (auto split: list.splits llist.splits) (transfer_prover)

lconcat lxs = case lxs of LNil ⇒ LNil | LCons xs lxs' ⇒ 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 ⇒ 'a llist ⇒ 'a llist where
  lapp lxs lys = case lxs of LNil ⇒ lys | LCons x lxs' ⇒ 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
    [] ⇒ (case lxs of LNil ⇒ LNil | LCons x lxs' ⇒ LCons x lxs')
  | x#xs' ⇒ LCons x (xs' @@ lxs))
  by (auto split: list.splits llist.splits) (transfer_prover)

lconcat lxs = case lxs of LNil ⇒ LNil | LCons xs lxs' ⇒ 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 ⇒ '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

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

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:
    $\textbf{codatatype}\ ('o, 'i)\ op = \text{Logic}\ (\text{apply}: ('i \Rightarrow ('o, 'i)\ op \times 'o\ list))$
  - 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

```
definition produce1' op lxs = while_option
  (λ(op, lxs). ¬ lnull lxs ∧ snd (apply op (lhd lxs)) = [])
  (λ(op, lxs). (fst (apply op (lhd lxs)), ltl lxs)) (op, lxs)
definition produce1 op lxs =
  (case produce1' op lxs of None ⇒ None
  | Some (op', lxs') ⇒ if lnull lxs' then None else
    let (op'', out) = apply op' (lhd lxs') in Some (op'', hd out, tl out, ltl lxs'))
corec produce where
  produce op lxs = (case produce1 op lxs of None ⇒ LNil
    | Some (op', x, xs, lxs') ⇒ LCons x (xs @@ produce op' lxs'))
```

- 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

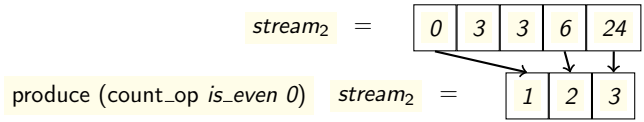
```
definition produce1' op lxs = while_option
  (λ(op, lxs). ¬ lnull lxs ∧ snd (apply op (lhd lxs)) = [])
  (λ(op, lxs). (fst (apply op (lhd lxs)), ltl lxs)) (op, lxs)
definition produce1 op lxs =
  (case produce1' op lxs of None ⇒ None
  | Some (op', lxs') ⇒ if lnull lxs' then None else
    let (op'', out) = apply op' (lhd lxs') in Some (op'', hd out, tl out, ltl lxs'))
corec produce where
  produce op lxs = (case produce1 op lxs of None ⇒ LNil
    | Some (op', x, xs, lxs') ⇒ LCons x (xs @@ produce op' lxs'))
```

- `produce1` 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 = fold (λe (op, out).
  let (op', out') = apply op e in (op', out @ out')) xs (op, [])
corec comp_op where
  comp_op op1 op2 = Logic (λev.
    let (op1', out) = apply op1 ev;  (op2', out') = fproduce op2 out
    in (comp_op op1' op2', 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 (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:

  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 (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:

  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))

  - Correctness: 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))

  - Correctness: Coinduction up to congruence for lazy list equality

Time-Aware Operators

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

Time-Aware Streams

- Time-Aware lazy lists

  $\textbf{datatype}$ ($'t$::*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 ⇒ 't set ⇒ 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* ⇒ 't set ⇒ *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	⋯

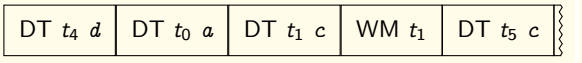
Monotone Time-Aware Streams

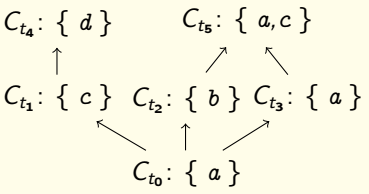
- Monotone: watermarks do not go back in time

coinductive monotone :: ('t::order, 'd) *event llist* ⇒ 't set ⇒ *bool* **where**
  LNil: monotone LNil *W*
| LConsR: (∀*wm'* ∈ *W*. ¬ *wm'* ≥ *wm*) ⟶ monotone *lxs* ({*wm*} ∪ *W*) ⟶
  monotone (LCons (WM *wm*) *lxs*) *W*
| LConsL: (∀*wm* ∈ *W*. ¬ *wm* ≥ *t*) ⟶ monotone *lxs W* ⟶
  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

$$
\begin{aligned}
&\textbf{coinductive}\ \text{productive}\ \textbf{where}\\
&\quad \text{LFinite}\mathit{:}\ \text{lfinite}\ \mathit{lxs} \longrightarrow \text{productive}\ \mathit{lxs}\\
&\mid \text{EnvWM}\mathit{:}\ \neg\ \text{lfinite}\ \mathit{lxs} \longrightarrow (\exists u \in \text{vimage WM (lset}\ \mathit{lxs}).\ u \geq t) \longrightarrow\\
&\quad \text{productive}\ \mathit{lxs} \longrightarrow \text{productive (LCons (DT}\ t\ d)\ \mathit{lxs})\\
&\mid \mathit{SkipWM}\mathit{:}\ \neg\ \text{lfinite}\ \mathit{lxs} \longrightarrow \text{productive}\ \mathit{lxs} \longrightarrow\\
&\quad \text{productive (LCons (WM}\ t)\ \mathit{lxs})
\end{aligned}
$$

  - 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

$$
\begin{aligned}
&\textbf{coinductive}\ \text{productive}\ \textbf{where}\\
&\quad \textit{LFinite:}\ \text{lfinite}\ \textit{lxs} \longrightarrow \text{productive}\ \textit{lxs}\\
&\quad |\ \textit{EnvWM:}\ \neg\ \text{lfinite}\ \textit{lxs} \longrightarrow (\exists u \in \text{vimage WM (lset}\ \textit{lxs}).\ u \geq t) \longrightarrow\\
&\qquad \text{productive}\ \textit{lxs} \longrightarrow \text{productive (LCons (DT}\ t\ d)\ \textit{lxs})\\
&\quad |\ \textit{SkipWM:}\ \neg\ \text{lfinite}\ \textit{lxs} \longrightarrow \text{productive}\ \textit{lxs} \longrightarrow\\
&\qquad \text{productive (LCons (WM}\ t)\ \textit{lxs})
\end{aligned}
$$

  - 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

Batch Operator: Soundness

Batch Operator: Completeness

- Uses soundness of batch_op
- Proof by induction over n

$$\begin{aligned}&\text{mono\_prod } lxs\ W \longrightarrow (\exists i\ d.\ \text{enat } i < \text{llength } lxs \wedge \text{lnth } lxs\ i = \text{DT } t\ d \wedge n = \text{Suc } i) \vee \\ &n = 0 \wedge t \in \text{set\_t } buf \longrightarrow (\forall t' \in \text{set\_t } buf.\ \text{lfinite } lxs \vee \exists wm \geq t'\ .\ \text{WM } wm \in \text{lset } lxs) \longrightarrow \\ &\exists wm\ batch.\ \text{DT } wm\ batch \in \text{lset } (\text{produce } (\text{batch\_op } buf)\ lxs) \wedge t \in \text{set\_t } batch \vee \\ &(\forall k \in \{n\ ..<\ \text{the\_enat } (\text{llength } lxs)\}\ .\ \neg\ (\exists t' \geq t.\ \text{lnth } lxs\ k = \text{WM } t')) \wedge \text{lfinite } lxs\end{aligned} \tag{1}$$

Batch Operator: Monotone

Batch Operator: Productive

Building Blocks: Incremental Operator

Batch Operator: Soundness

Batch Operator: Completeness

Batch Operator: Monotone

Batch Operator: Productive

Compositional Reasoning

Case Study

Histogram

Histogram: Soundness

Histogram: Completeness

Histogram: Monotone

Histogram: Productive

Efficient Histogram

- Foo

Join

Join: Soundness

Join: Completeness

Join: Monotone

Next Steps

Next Steps

Questions, comments and suggestions