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

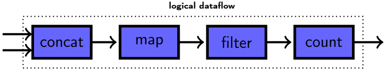
Stream Processing

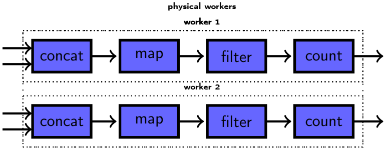
- Stream Processing: Abstraction for processing data when the input is not completely presented in the begging of the computation
- Dataflow Model:
  - Directed graph of interconnected operators that perform event-wise transformations
  - Examples: Apache Flink, Apache Samza, Apache Spark, Google Cloud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

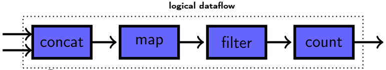


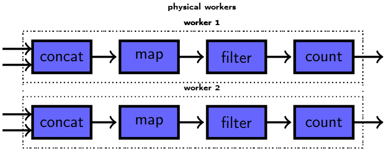


- Time-Aware Computations
  - Timestamps: Metadata associating the data with some data collection
  - Watermarks: Metadata indicating the completion of a data collection

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

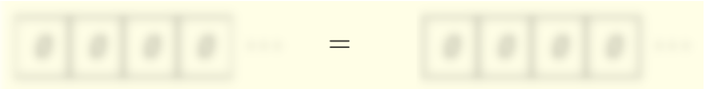
- Examples:
  - LNil
  - LCons *1* (LCons *2* (LCons *3* LNil))
  - LCons *0* (LCons *0* (LCons *0* (. . .)))
- Induction principle assuming membership in the lazy list
- Coinductive principle for lazy list equality:
  - Show that there is a pair of goggles that makes them to look the same, which implies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

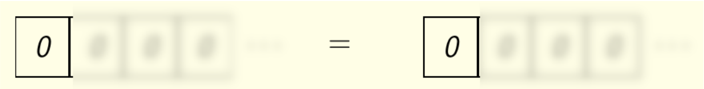


Isabelle/HOL: (Co)datatypes

- 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

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

Isabelle/HOL: Recursion and While Combinator

- Recursion

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 $b\ c\ s = \ldots$

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

Isabelle/HOL: Corecursion and Friends

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

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

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

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

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

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

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

$\textbf{corec}$ lapp :: $'a$ $llist \Rightarrow 'a$ $llist \Rightarrow 'a$ $llist$ $\textbf{where}$
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.

$\textbf{friend\_of\_corec}$ lshift $\textbf{where}$
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))
$\textbf{by}$ (*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:

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

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

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

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

Isabelle/HOL: Corecursion and Friends

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

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

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

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

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

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

Isabelle/HOL: (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

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 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 $\mathsf{produce_1}'$ *op lxs* = while_option
  (λ(*op*, *lxs*). $\neg$ lnull *lxs* $\wedge$ snd (apply *op* (lhd *lxs*)) = [])
  (λ(*op*, *lxs*). (fst (apply *op* (lhd *lxs*)), ltl *lxs*)) (*op*, *lxs*)
definition $\mathsf{produce_1}$ *op lxs* =
  (`case` $\mathsf{produce_1}'$ *op lxs* `of` None $\Rightarrow$ None
  | Some (*op'*, *lxs'*) $\Rightarrow$ `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` $\mathsf{produce_1}$ *op lxs* `of` None $\Rightarrow$ LNil
    | Some (*op'*, *x*, *xs*, *lxs'*) $\Rightarrow$ LCons *x* (*xs* @@ produce *op'* *lxs'*))

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

Execution formalization

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

$$
\begin{array}{l}
\textbf{definition}\ \mathrm{produce}_1{}'\ \mathit{op}\ \mathit{lxs} = \mathrm{while\_option} \\
\quad (\lambda(\mathit{op}, \mathit{lxs}).\ \neg\ \mathrm{lnull}\ \mathit{lxs} \wedge \mathrm{snd}\ (\mathrm{apply}\ \mathit{op}\ (\mathrm{lhd}\ \mathit{lxs})) = []) \\
\quad (\lambda(\mathit{op}, \mathit{lxs}).\ (\mathrm{fst}\ (\mathrm{apply}\ \mathit{op}\ (\mathrm{lhd}\ \mathit{lxs})), \mathrm{ltl}\ \mathit{lxs}))\ (\mathit{op}, \mathit{lxs}) \\
\textbf{definition}\ \mathrm{produce}_1\ \mathit{op}\ \mathit{lxs} = \\
\quad (\mathtt{case}\ \mathrm{produce}_1{}'\ \mathit{op}\ \mathit{lxs}\ \mathtt{of}\ \mathrm{None} \Rightarrow \mathrm{None} \\
\quad \mid \mathrm{Some}\ (\mathit{op}', \mathit{lxs}') \Rightarrow \mathtt{if}\ \mathrm{lnull}\ \mathit{lxs}'\ \mathtt{then}\ \mathrm{None}\ \mathtt{else} \\
\quad\quad \mathtt{let}\ (\mathit{op}'', \mathit{out}) = \mathrm{apply}\ \mathit{op}'\ (\mathrm{lhd}\ \mathit{lxs}')\ \mathtt{in}\ \mathrm{Some}\ (\mathit{op}'', \mathrm{hd}\ \mathit{out}, \mathrm{tl}\ \mathit{out}, \mathrm{ltl}\ \mathit{lxs}')) \\
\textbf{corec}\ \mathrm{produce}\ \textbf{where} \\
\quad \mathrm{produce}\ \mathit{op}\ \mathit{lxs} = (\mathtt{case}\ \mathrm{produce}_1\ \mathit{op}\ \mathit{lxs}\ \mathtt{of}\ \mathrm{None} \Rightarrow \mathrm{LNil} \\
\quad\quad \mid \mathrm{Some}\ (\mathit{op}', x, \mathit{xs}, \mathit{lxs}') \Rightarrow \mathrm{LCons}\ x\ (\mathit{xs}\ @@\ \mathrm{produce}\ \mathit{op}'\ \mathit{lxs}'))
\end{array}
$$

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

- Example:

$$\textbf{corec}\ \mathsf{count\_op}\ \textbf{where}\ \mathsf{count\_op}\ P\ n =$$
$$\quad \mathsf{Logic}\ (\lambda e.\ \texttt{if}\ P\ e\ \texttt{then}\ (\mathsf{count\_op}\ P\ (n+1),\ [n{+}1])\ \texttt{else}\ (\mathsf{count\_op}\ P\ n,\ []))$$

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

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

Sequential Composition

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

$$
\begin{array}{l}
\textbf{definition}\ \text{fproduce}\ \mathit{op}\ \mathit{xs} = \text{fold}\ (\lambda e\ (\mathit{op}, \mathit{out}).\\
\quad \texttt{let}\ (\mathit{op}', \mathit{out}') = \text{apply}\ \mathit{op}\ e\ \texttt{in}\ (\mathit{op}', \mathit{out}\ @\ \mathit{out}'))\ \mathit{xs}\ (\mathit{op}, [])\\
\textbf{corec}\ \text{comp\_op}\ \textbf{where}\\
\quad \text{comp\_op}\ \mathit{op}_1\ \mathit{op}_2 = \text{Logic}\ (\lambda \mathit{ev}.\\
\qquad \texttt{let}\ (\mathit{op}_1', \mathit{out}) = \text{apply}\ \mathit{op}_1\ \mathit{ev};\ (\mathit{op}_2', \mathit{out}') = \text{fproduce}\ \mathit{op}_2\ \mathit{out}\\
\qquad \texttt{in}\ (\text{comp\_op}\ \mathit{op}_1'\ \mathit{op}_2', \mathit{out}'))
\end{array}
$$

Sequential Composition: Correctness

- Correctness:

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

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

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

  - Correctness: Coinduction up to congruence for lazy list equality

Sequential Composition: Correctness

- Correctness:

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

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

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

  - Correctness: Coinduction up to congruence for lazy list equality

Sequential Composition: Correctness

- Correctness:

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

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

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

  - Correctness: Coinduction up to congruence for lazy list equality

Sequential Composition: Correctness

- Correctness:

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

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

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

  - Correctness: Coinduction up to congruence for lazy list equality

Time-Aware Operators

Time-Aware Streams

- Time-Aware lazy lists

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

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

Time-Aware Streams

- Time-Aware lazy lists

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

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

Monotone Time-Aware Streams

- Monotone: watermarks do not go back in time

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

- Up to congruence coinduction principle
- Example:

$stream_3 =$

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

...

Monotone Time-Aware Streams

- Monotone: watermarks do not go back in time

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

- Up to congruence coinduction principle
- Example:

$stream_3 =$

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

Monotone Time-Aware Streams

- Monotone: watermarks do not go back in time

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

- Up to congruence coinduction principle
- Example:

$stream_3$ =

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

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

Productive Time-Aware Streams

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

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

  - Up to congruence coinduction principle

Productive Time-Aware Streams

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

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

  - 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

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

  - Up to congruence coinduction principle

Building Blocks: Batch Operator

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

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

Building Blocks: Batch Operator

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

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

Batch Operator: Soundness

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

$$\mathsf{produce}\ (\mathsf{batch\_op}\ [(t_0,a)])\ stream_3 \; = \; \boxed{\mathsf{DT}\ t_1\ [(t_0,a),(t_0,a),(t_1,c)]} \boxed{\mathsf{WM}\ t_1} \boxed{\mathsf{DT}\ t_2\ [(t_2,b)]} \boxed{\mathsf{WM}\ t_2} \boxed{\mathsf{DT}\ t_3\ [(t_3,a)]} \boxed{\mathsf{DT}\ t_5\ [(t_5,a),(t_5,c)]} \boxed{\mathsf{WM}\ t_5} \cdots$$

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