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

Introduction

Stream Processing

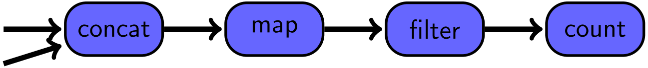
- Stream Processing: Abstraction for processing data when the input is not completely presented in the begging of the computation
- Dataflow Model:
  - Directed graph of interconnected operators that perform event-wise transformations
  - Examples: Apache Flink, Apache Samza, Apache Spark, Google Cloud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

Stream Processing

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

Cloud DataFlow Flink

  - Highly Parallel



Preliminaries

Isabelle/HOL

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

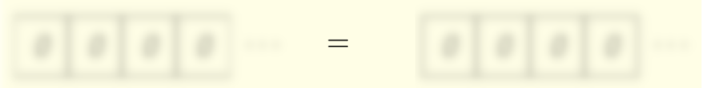
- Isabelle/HOL: Isabelle's flavor of HOL
- All functions in Isabelle/HOL must be total

Isabelle/HOL: (Co)datatypes

- Datatypes and Codatatypes

$\textbf{codatatype}$ (lset: $'a$) *llist* = lnull: LNil | LCons (lhd: $'a$) (ltl: $'a$ *llist*)
  $\textbf{for}$ map: lmap $\textbf{where}$ 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 if empty iff second is
    - They have the same head
    - Their tail looks the same

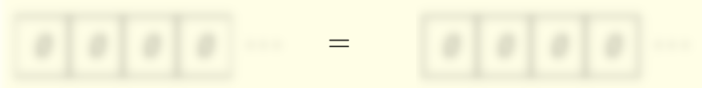


Isabelle/HOL: (Co)datatypes

- 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 if 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 if empty iff second is
    - They have the same head
    - Their tail looks the same

$$\boxed{0} \; = \; \boxed{0}$$

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 if empty iff second is
    - They have the same head
    - Their tail looks the same

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

Isabelle/HOL: Recursion and While Combinator

- Recursion

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

- While Combinator

$\textbf{definition}$ while_option $::\ ('a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a\ option$ $\textbf{where}$
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:

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:

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

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

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

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

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

Isabelle/HOL: Corecursion and Friends

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

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

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

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

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

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

Isabelle/HOL: Corecursion and Friends

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

corec lapp :: 'a llist ⇒ '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 $:: 'a \Rightarrow 'a\ llist \Rightarrow bool$ **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

Isabelle/HOL: (Co)inductive Predicates

- Inductive predicate
  - Finite number of introduction rule applications

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

$\mathsf{in\_llist}\ \mathit{2}\ (\mathsf{LCons}\ \mathit{1}\ (\mathsf{LCons}\ (\mathit{2}\ (\ldots))))$

- Coinductive predicate
  - Infinite number of introduction rule applications

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

$\mathsf{lprefix}\ (\mathsf{LCons}\ \mathit{1}\ (\mathsf{LCons}\ (\mathit{2}\ (\ldots))))\ (\mathsf{LCons}\ \mathit{1}\ (\mathsf{LCons}\ (\mathit{2}\ (\ldots))))$

- Coinduction principle

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

Lazy Lists Processors

Operator formalization

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

Operator formalization

- Operator as a codatatype
  - Taking $'i$ as the input type, and $'o$ as the output type:
    $\textbf{codatatype}\ ('o, 'i)\ \mathit{op} = \mathrm{Logic}\ (\mathrm{apply}\!: ('i \Rightarrow ('o, 'i)\ \mathit{op} \times 'o\ \mathit{list}))$
  - 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
  $(\lambda(\mathit{op}, \mathit{lxs}).\ \neg\ \mathsf{lnull}\ \mathit{lxs} \wedge \mathsf{snd}\ (\mathsf{apply}\ \mathit{op}\ (\mathsf{lhd}\ \mathit{lxs})) = [])$
  $(\lambda(\mathit{op}, \mathit{lxs}).\ (\mathsf{fst}\ (\mathsf{apply}\ \mathit{op}\ (\mathsf{lhd}\ \mathit{lxs})), \mathsf{ltl}\ \mathit{lxs}))\ (\mathit{op}, \mathit{lxs})$
definition $\mathsf{produce}_1$ op lxs =
  $(\mathtt{case}\ \mathsf{produce}_1{}'\ \mathit{op}\ \mathit{lxs}\ \mathtt{of}\ \mathsf{None} \Rightarrow \mathsf{None}$
  $\mid \mathsf{Some}\ (\mathit{op}', \mathit{lxs}') \Rightarrow \mathtt{if}\ \mathsf{lnull}\ \mathit{lxs}'\ \mathtt{then}\ \mathsf{None}\ \mathtt{else}$
    $\mathtt{let}\ (\mathit{op}'', \mathit{out}) = \mathsf{apply}\ \mathit{op}'\ (\mathsf{lhd}\ \mathit{lxs}')\ \mathtt{in}\ \mathsf{Some}\ (\mathit{op}'', \mathsf{hd}\ \mathit{out}, \mathsf{tl}\ \mathit{out}, \mathsf{ltl}\ \mathit{lxs}'))$
corec produce **where**
  $\mathsf{produce}\ \mathit{op}\ \mathit{lxs} = (\mathtt{case}\ \mathsf{produce}_1\ \mathit{op}\ \mathit{lxs}\ \mathtt{of}\ \mathsf{None} \Rightarrow \mathsf{LNil}$
    $\mid \mathsf{Some}\ (\mathit{op}', x, \mathit{xs}, \mathit{lxs}') \Rightarrow \mathsf{LCons}\ x\ (\mathit{xs}\ @@\ \mathsf{produce}\ \mathit{op}'\ \mathit{lxs}'))$

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

Execution formalization

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

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

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

Operators: Count

- Example:

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

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

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

Sequential Composition

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

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

- 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:

$\text{produce}\ (\text{comp\_op}\ \mathit{op}_1\ \mathit{op}_2)\ \mathit{lxs} = \text{produce}\ \mathit{op}_2\ (\text{produce}\ \mathit{op}_1\ \mathit{lxs})$

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

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

  - 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

Sequential Composition: Correctness

- Correctness:

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

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

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

  - Correctness: Coinduction up to congruence for lazy list equality

Time-Aware Operators

Time-Aware Streams

- Time-Aware lazy lists

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

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

Time-Aware Streams

- Time-Aware lazy lists

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

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

Monotone Time-Aware Streams

- Monotone: watermarks do not go back in time

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

- Up to congruence coinduction principle
- Example:

$stream_3 =$

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

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

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 …

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

Productive Time-Aware Streams

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

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

  - Up to congruence coinduction principle

Productive Time-Aware Streams

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

```
coinductive productive where
  LFinite: lfinite lxs ⟶ 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: 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

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

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