Infinite Isabelle-LLVM

Rafael Castro G. Silva

rasi@di.ku.dk

Department of Computer Science
University of Copenhagen

02/11/2023

What my PhD is about?

What my PhD is about?

- Distributed Systems
  - Stream processing frameworks
    - Dataflow models
      - Time-Aware Computations

What my PhD is about?

- Distributed Systems
  - Stream processing frameworks
    - Dataflow models
      - Time-Aware Computations
- Formal Methods
  - Verification using proof assistants
    - Isabelle proofs
      - Verified + executable + efficient code
- Formalization of Time-Aware Stream Processing

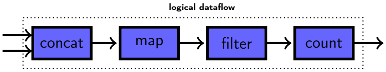
Introduction

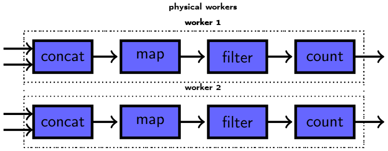
Stream Processing

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

Stream Processing

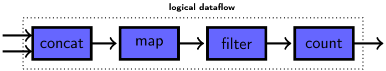
- Stream Processing: Abstraction for processing data when the input is not completely presented in the begging of the computation
- Dataflow Model:
    - Directed graph of interconnected operators that perform event-wise transformations
    - E.g.: Apache Flink, Apache Samza, Apache Spark, Google Cloud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

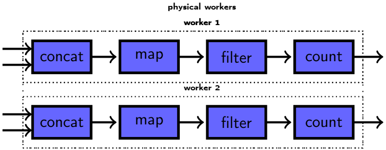




Stream Processing

- 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
  - E.g.: 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/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* (...)))

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

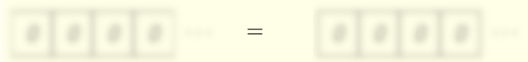
- Examples:
  - LNil
  - LCons *1* (LCons *2* (LCons *3* LNil))
  - LCons *0* (LCons *0* (LCons *0* (...)))
- Induction principle with lset assumption
  - If $x \in$ lset *lxs*, and if P holds for all lazy lists containing x, then P *lxs* is true

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 with lset assumption
  - If $x \in$ lset *lxs*, and if P holds for all lazy lists containing x, then P *lxs* is true
- Coinductive principle for lazy list equality:
  - Show that there is a "pair of goggles" (relation) that makes them to look the same:
    - The first lazy list is empty iff second is
    - They have the same head
    - Their tail looks the same

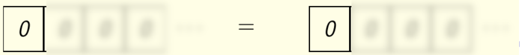


Isabelle/HOL: (Co)datatypes

- 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 with lset assumption
  - If $x \in$ lset *lxs*, and if P holds for all lazy lists containing x, then P *lxs* is true
- Coinductive principle for lazy list equality:
  - Show that there is a "pair of goggles " (relation) that makes them to look the same:
    - 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 with lset assumption
  - If $x \in$ lset *lxs*, and if *P* holds for all lazy lists containing *x*, then *P lxs* is true
- Coinductive principle for lazy list equality:
  - Show that there is a "pair of goggles " (relation) that makes them to look the same:
    - The first lazy list is empty iff second is
    - They have the same head
    - Their tail looks the same

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

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* = ...

- 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

Isabelle/HOL: Corecursion and Friends

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

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

Isabelle/HOL: Corecursion and Friends

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

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

- Friendly function
  - Preserves productivity: it may consume at most one constructor to produce one constructor.
  - lshift (@@) is proved to a be friend

Isabelle/HOL: Corecursion and Friends

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

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

- Friendly function
  - Preserves productivity: it may consume at most one constructor to produce one constructor.
  - lshift (@@) is proved to a be friend
- 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* (. . .))))

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 ($\ldots$))))

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:
    $\mathbf{codatatype}\ ('o, 'i)\ op = \mathrm{Logic}\ (\mathrm{apply}\colon ('i \Rightarrow ('o, 'i)\ op \times 'o\ list))$

Operator formalization

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

Execution formalization

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

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

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

Execution formalization

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

  definition $\mathsf{produce}_1$ *op lxs* = while_option ...

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

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

Operators: Count

- Example:

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

$$\mathit{stream}_1 \quad = \quad \boxed{0}\,\boxed{3}\,\boxed{3}\,\boxed{6}\,\boxed{24}$$

$$\mathsf{produce}\ (\mathsf{count\_op}\ \mathit{is\_even}\ 0)\quad \mathit{stream}_1 \quad = \quad \boxed{1}\,\boxed{2}\,\boxed{3}$$

Sequential Composition

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

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

Sequential Composition: Correctness

- Correctness:

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

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

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

$$\text{produce } (\text{skip\_op } op \; n) \; lxs = \text{ldropn } n \; (\text{produce } op \; lxs)$$

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

    $$\text{produce } (\text{skip\_op } op \ n) \ lxs = \text{ldropn } n \ (\text{produce } op \ lxs)$$

  - Proof: Coinduction up to congruence for lazy list equality

Time-Aware Operators

Time-Aware Streams

- Time-Aware lazy lists

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

Time-Aware Streams

- Time-Aware lazy lists

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

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

Monotone Time-Aware Streams

- Monotone: watermarks do not go back in time

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

- Up to congruence coinduction principle
- Example:

$stream_2 =$

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

…

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

Productive Time-Aware Streams

- Productive: always eventually allows the production

Productive Time-Aware Streams

- Productive: always eventually allows the production
  - Batching operators: accumulate data until its completion

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*)

stream$_2$ =

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

...

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

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*)

stream$_2$ =

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

WM t_2	DT t_5 c	DT t_3 a	DT t_5 a	WM t_5

...

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

- Up to congruence coinduction principle

Building Blocks: Batch Operator

- Building Blocks: reusable operators
  - Batching and incremental computations

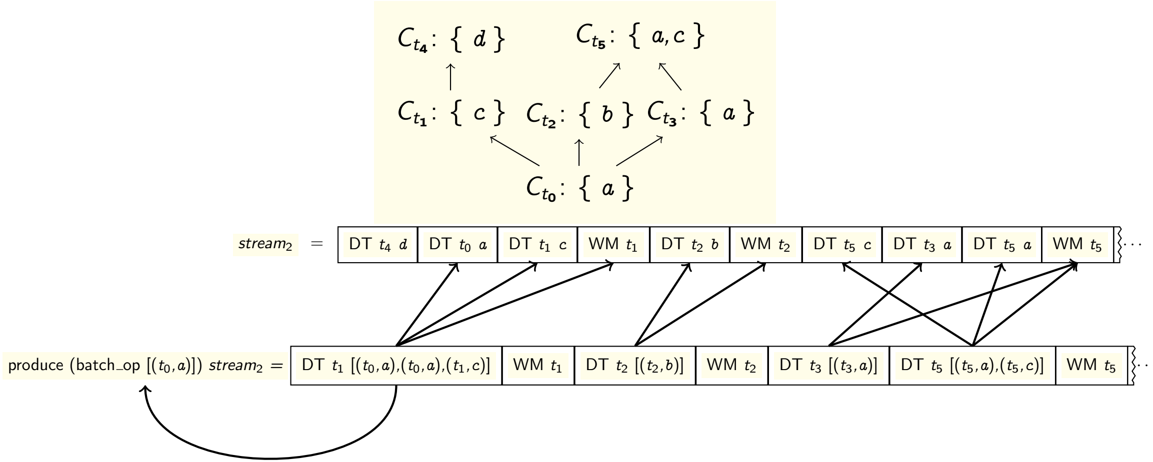
Building Blocks: Batch Operator

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

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

Batch Operator: Soundness

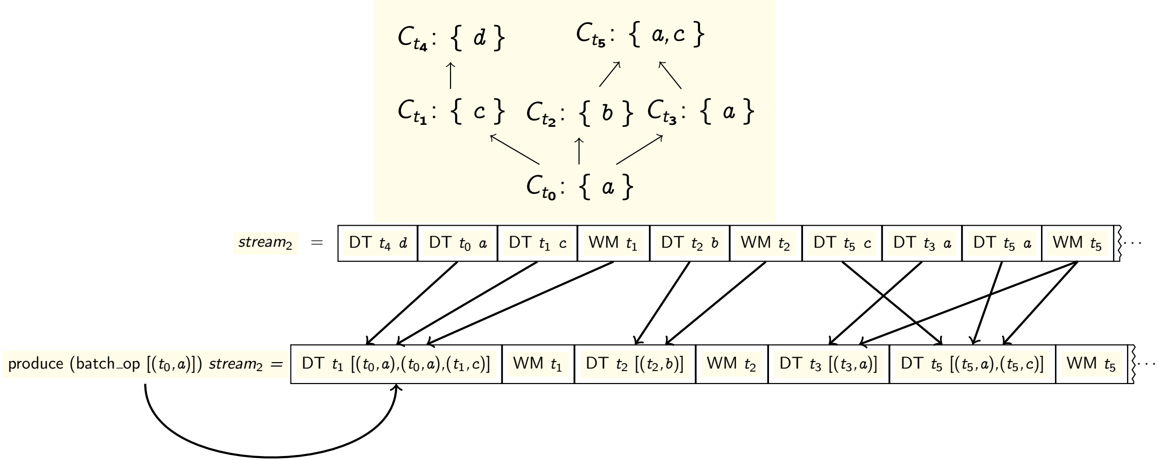
- Given a monotone time-aware stream



- Proof: lset induction, produce_1 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!

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

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

- Proof: coinduction up to congruence

Building Blocks: Incremental Operator

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

$$
\begin{array}{l}
\textbf{corec}\ \mathsf{incr\_op}\ \textbf{where} \\
\quad \mathsf{incr\_op}\ \mathit{buf} = \mathsf{Logic}\ (\lambda\ \mathit{ev}.\ \mathtt{case}\ \mathit{ev}\ \mathtt{of}\ \mathsf{DT}\ \mathit{wm}\ \mathit{batch} \Rightarrow \\
\qquad \mathtt{let}\ \mathit{out} = \mathsf{map}\ (\lambda t.\ \mathsf{DT}\ t\ (\mathit{buf}\ @\ \mathit{batch}))\ (\mathsf{remdups}\ (\mathsf{map}\ \mathsf{fst}\ \mathit{batch})) \\
\qquad \mathtt{in}\ (\mathsf{incr\_op}\ (\mathit{buf}\ @\ \mathit{batch}), \mathit{out}) \\
\quad \mid \mathsf{WM}\ \mathit{wm} \Rightarrow\ (\mathsf{incr\_op}\ \mathit{buf}, [\mathsf{WM}\ \mathit{wm}]))
\end{array}
$$

Incremental Operator: Soundness

$stream_2$ = | DT t_4 d | DT t_0 a | DT t_1 c | WM t_1 | DT t_2 b | WM t_2 | DT t_5 c | DT t_3 a | DT t_5 a | WM t_5 | ⋯

$stream_3$ = produce (batch_op $[(t_0, a)]$) $stream_2$ = | DT t_1 $[(t_0,a),(t_0,a),(t_1,c)]$ | WM t_1 | DT t_2 $[(t_2,b)]$ | WM t_2 | DT t_3 $[(t_3,a)]$ | DT t_5 $[(t_5,a),(t_5,c)]$ | WM t_5 | ⋯

produce (incr_op []) $stream_3$ = | DT t_0 $[(t_0,a),(t_0,a),(t_1,c)]$ | DT t_1 $[(t_0,a),(t_0,a),(t_1,c)]$ | WM t_1 | DT t_2 $[(t_0,a),(t_0,a),(t_1,c),(t_2,b)]$ | WM t_2 | DT t_3 $[(t_0,a),(t_0,a),(t_1,c),(t_2,b),(t_3,a)]$ | ⋯

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

Incremental Operator: Completeness

$stream_2$ = | DT t_4 d | DT t_0 a | DT t_1 c | WM t_1 | DT t_2 b | WM t_2 | DT t_5 c | DT t_3 a | DT t_5 a | WM t_5 | ⋯

$stream_3$ = produce (batch_op [(t_0,a)]) $stream_2$ = | DT t_1 [(t_0,a),(t_0,a),(t_1,c)] | WM t_1 | DT t_2 [(t_2,b)] | WM t_2 | DT t_3 [(t_3,a)] | DT t_5 [(t_5,a),(t_5,c)] | WM t_5 | ⋯

produce (incr_op []) $stream_3$ = | DT t_0 [(t_0,a),(t_0,a),(t_1,c)] | DT t_1 [(t_0,a),(t_0,a),(t_1,c)] | WM t_1 | DT t_2 [(t_0,a),(t_0,a),(t_1,c),(t_2,b)] | WM t_2 | DT t_3 [(t_0,a),(t_0,a),(t_1,c),(t_2,b),(t_3,a)] | ⋯

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

Incremental Operator: Monotone and productive preservation

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

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

- Proof: coinduction up to congruence

Compositional Reasoning

- batch_op and incr_op can be composed

  definition incr_batch_op *buf1 buf2* = comp_op (batch_op *buf1*) (incr_op *buf2*)

- Soundness, completeness, and monotone and productive preservation

Case Study

Histogram

- A histogram count the elements of a collection
- Incremental histogram: timestamps smaller or equal
- $H_{t_5} = C_{t_0} + C_{t_0} + C_{t_1} + C_{t_2} + C_{t_5}$
- paths to t_5: $\{t_0, t_2\}$ and $\{t_0, t_3\}$

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

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

Histogram Operator

$\textbf{corec}$ map_op $\textbf{where}$ map_op f = Logic (λ ev. case ev of
WM $wm \Rightarrow$ (map_op f, [WM wm]) | DT t $d \Rightarrow$ (map_op f, [DT t (f t d)]))

$\textbf{definition}$ incr_coll t xs = mset ...

$\textbf{definition}$ incr_hist_op $buf1$ $buf2$ =
comp_op (incr_batch_op $buf1$ $buf2$) (map_op incr_coll)

Histogram Operator: Correctness

- Correctness: (1) Soundness + (2) Completeness + (3) Monotone Preservation + (4) Productive Preservation
- Given a monotone and productive time-aware stream

$stream_2$ = | DT t_4 d | DT t_0 a | DT t_1 c | WM t_1 | DT t_2 b | WM t_2 | DT t_5 c | DT t_3 a | DT t_5 a | WM t_5 | $\cdots$

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

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

Efficient Histogram Operator

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

Join Operator

- Relation Join
- Use the `sum` type in the timestamps to represent two stream as one
- Partial order for the `sum`: lefts and rights are incomparable
- Defined using `incr_batch_op`
- Soundness, Completeness, Monotone
  - WIP: Productive

Next Steps

Next Steps

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

Questions, comments and suggestions