Time-Aware Stream Processing in Isabelle/HOL

Rafael Castro G. Silva and Dmitriy Traytel

rasi@di.ku.dk,traytel@di.ku.dk

Department of Computer Science
University of Copenhagen

14/09/2024

Introduction

Time-Aware Stream Processing

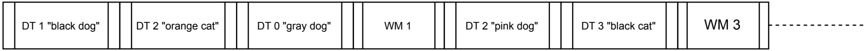
- What is Time-Aware Stream Processing?

Time-Aware Stream Processing

- What is Time-Aware Stream Processing?
  - Stream Processing: programs that compute (possibly) unbounded sequences of data (streams)

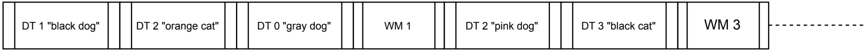
Time-Aware Stream Processing

- What is Time-Aware Stream Processing?
  - Stream Processing: programs that compute (possibly) unbounded sequences of data (streams)
  - Time-Aware: Data has timestamp metadata; timestamps are bounded by watermarks
    - Timestamp: A partially-ordered value associated with the data (e.g., unix-time, int, pairs of ints, etc...)
    - Watermark: A value of the same type of the timestamp. Represents data-completeness.



Time-Aware Stream Processing

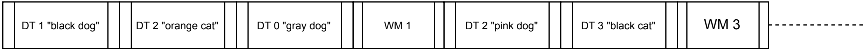
- What is Time-Aware Stream Processing?
  - Stream Processing: programs that compute (possibly) unbounded sequences of data (streams)
  - Time-Aware: Data has timestamp metadata; timestamps are bounded by watermarks
    - Timestamp: A partially-ordered value associated with the data (e.g., unix-time, int, pairs of ints, etc...)
    - Watermark: A value of the same type of the timestamp. Represents data-completeness.



- Asynchronous Dataflow Programming: Directed graph of interconnected operators that perform event-wise transformations
- E.g.: Apache Flink, Apache Samza, Apache Spark, Google Cloud Dataflow, and Timely Dataflow

Time-Aware Stream Processing

- What is Time-Aware Stream Processing?
  - Stream Processing: programs that compute (possibly) unbounded sequences of data (streams)
  - Time-Aware: Data has timestamp metadata; timestamps are bounded by watermarks
    - Timestamp: A partially-ordered value associated with the data (e.g., unix-time, int, pairs of ints, etc...)
    - Watermark: A value of the same type of the timestamp. Represents data-completeness.

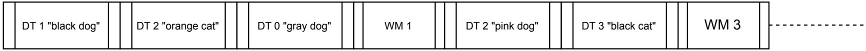


- Asynchronous Dataflow Programming: Directed graph of interconnected operators that perform event-wise transformations
- E.g.: Apache Flink, Apache Samza, Apache Spark, Google Cloud Dataflow, and Timely Dataflow

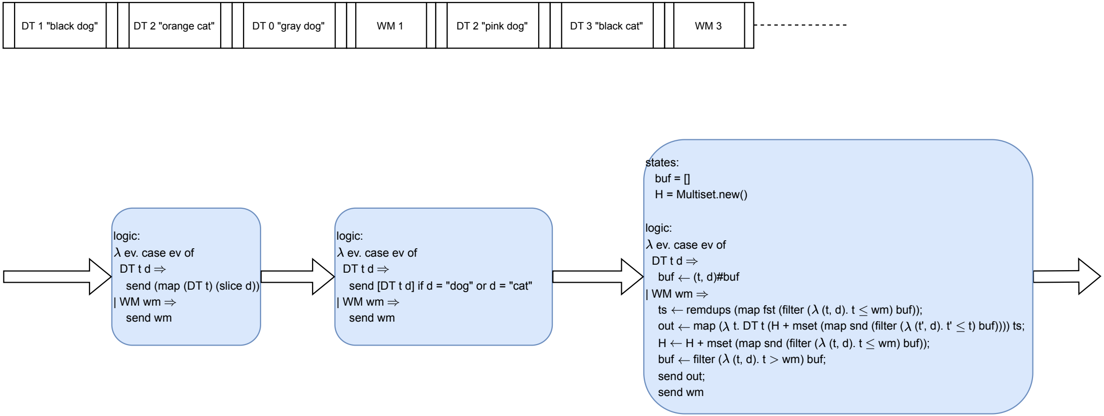
- Why?
  - Highly Parallel
  - Low latency (output as soon as possible)
  - Incremental computing (re-uses previous computations)

Time-Aware Stream Processing Example

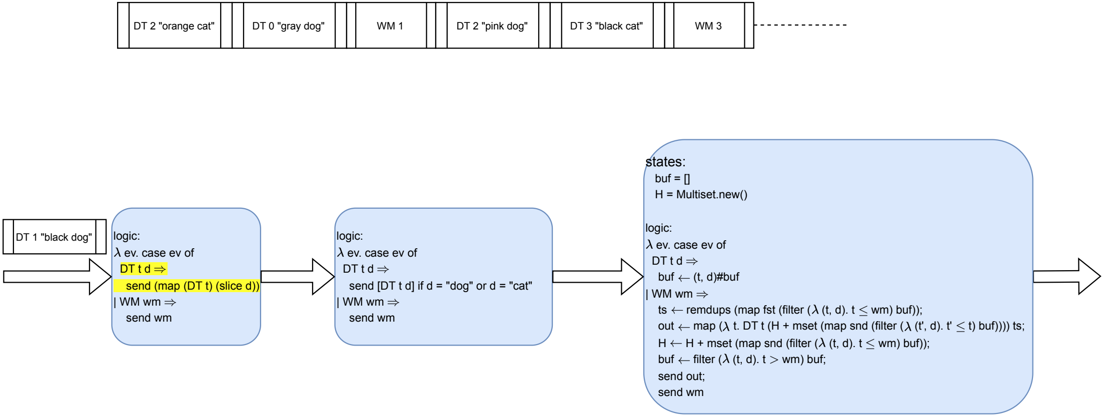
Example:
Incrementally count the occurrences of the words “dog” and “cat”



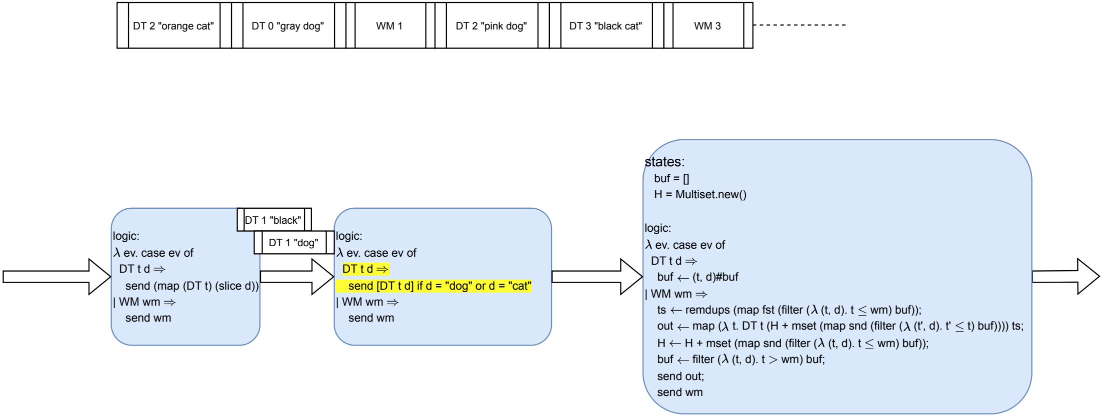
Time-Aware Stream Processing Example



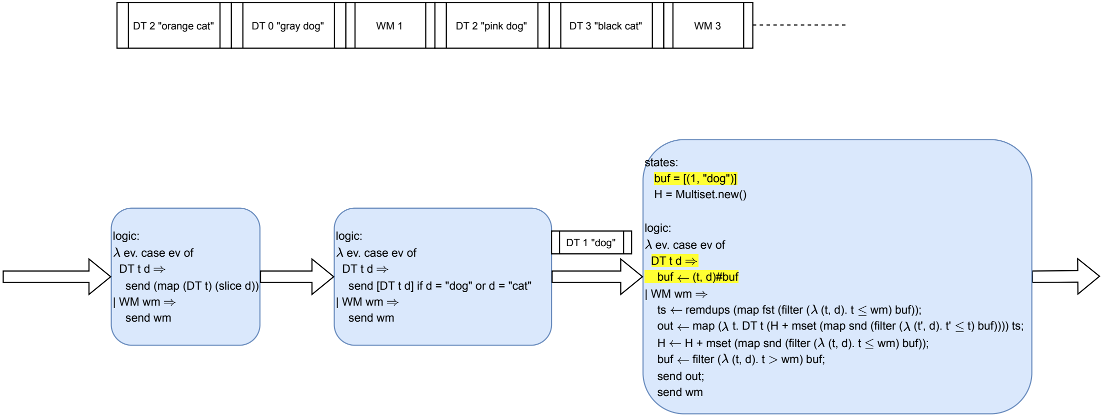
Time-Aware Stream Processing Example



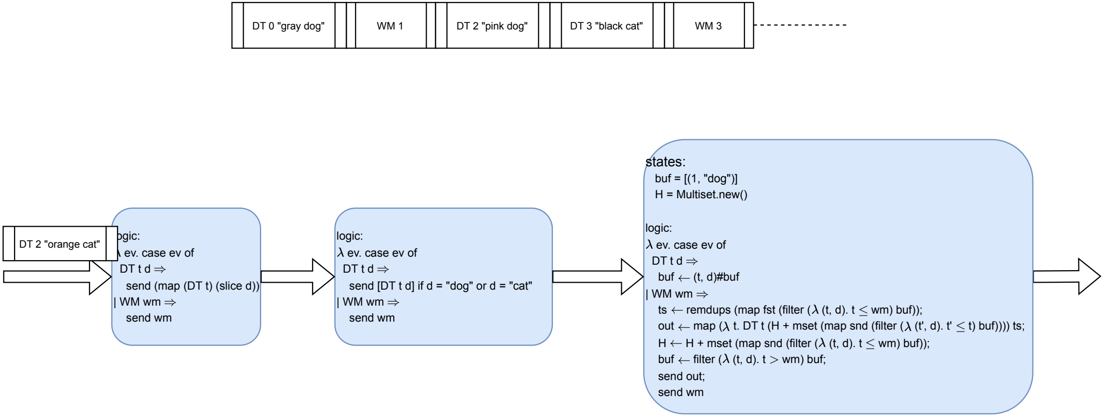
Time-Aware Stream Processing Example



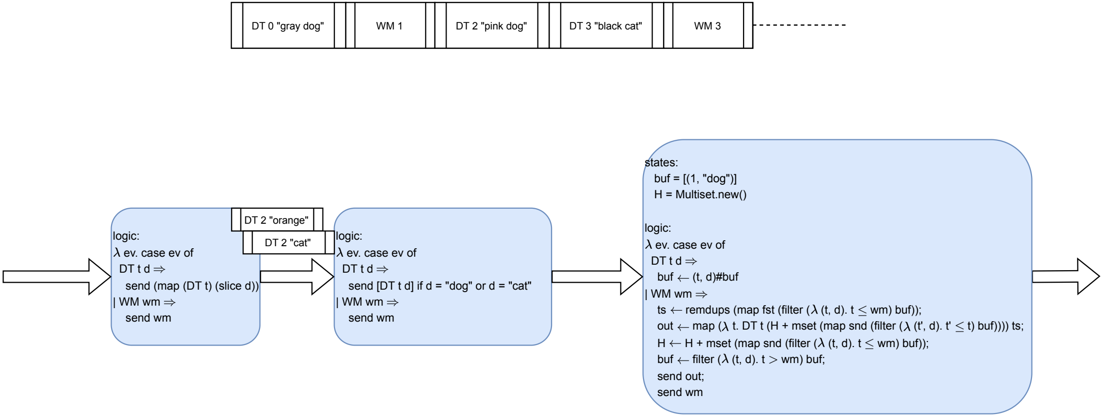
Time-Aware Stream Processing Example



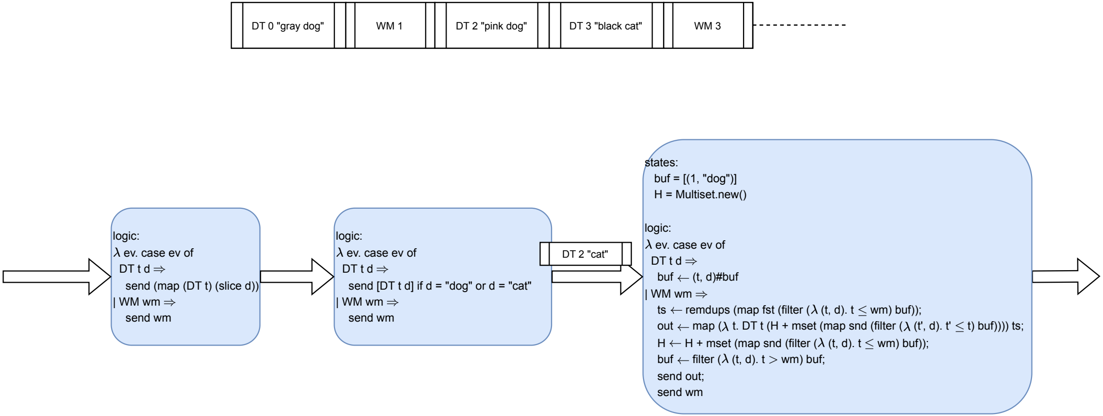
Time-Aware Stream Processing Example



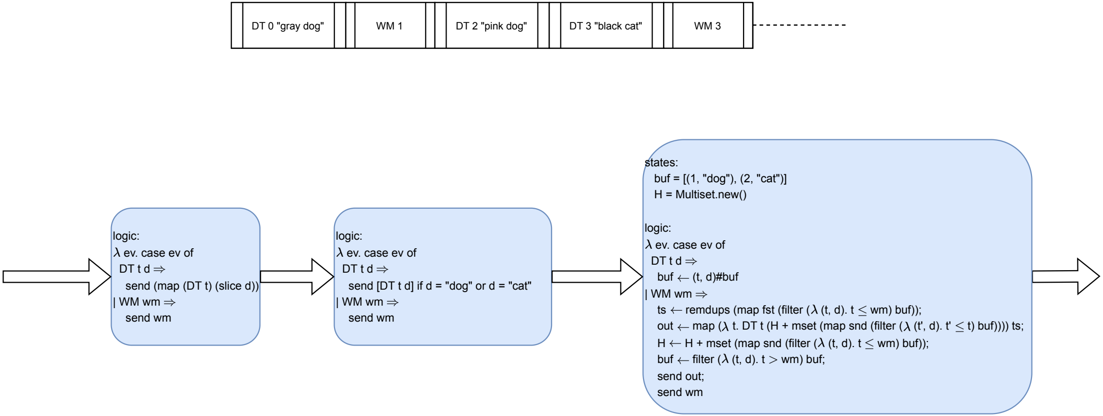
Time-Aware Stream Processing Example



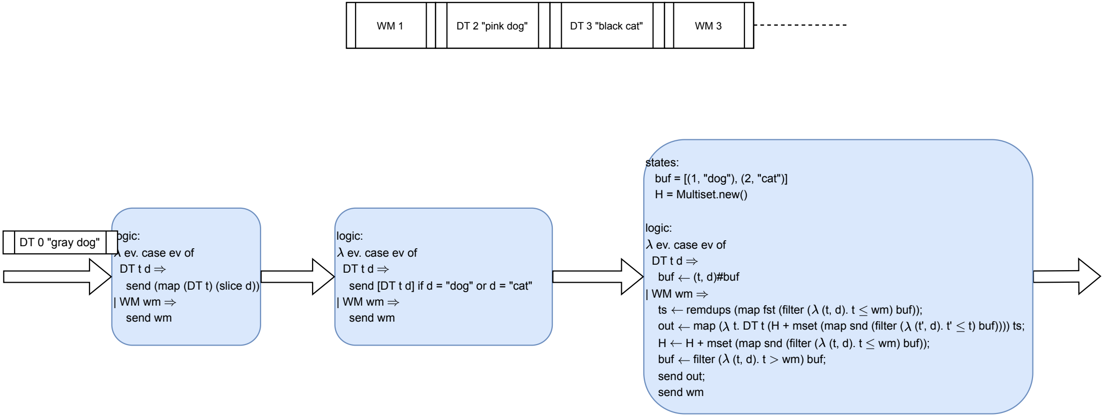
Time-Aware Stream Processing Example



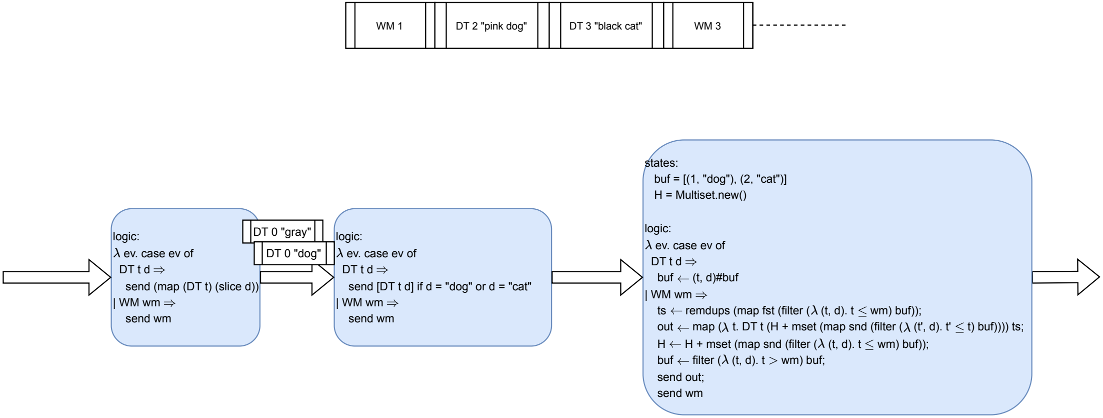
Time-Aware Stream Processing Example



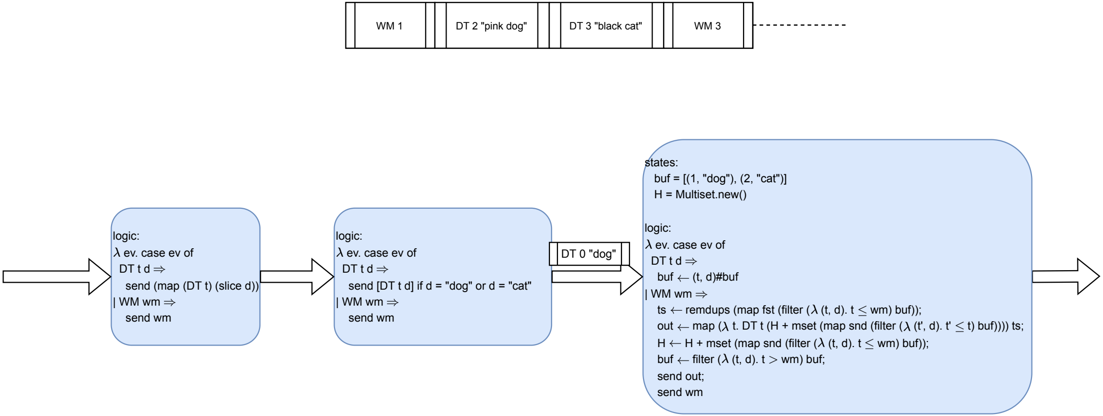
Time-Aware Stream Processing Example



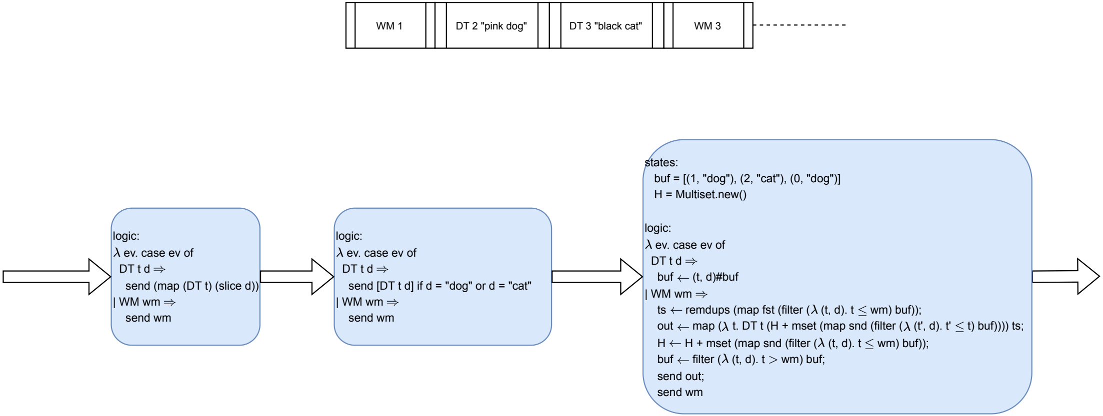
Time-Aware Stream Processing Example



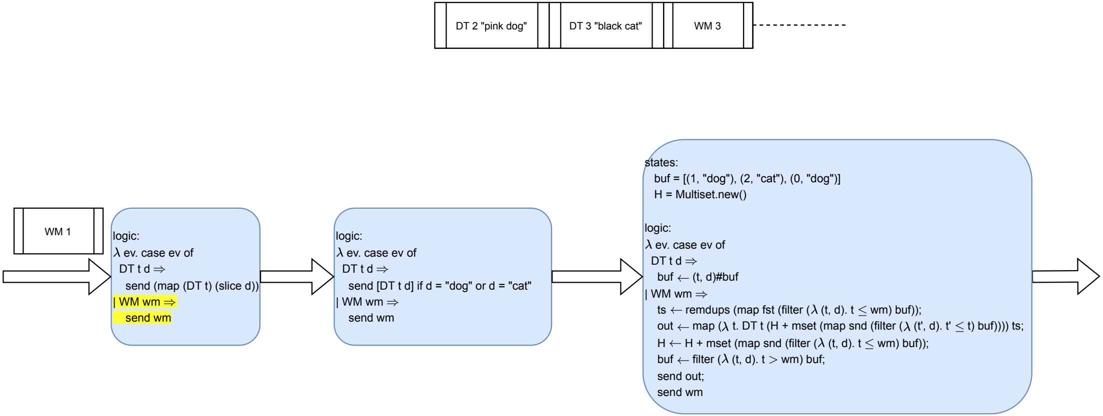
Time-Aware Stream Processing Example



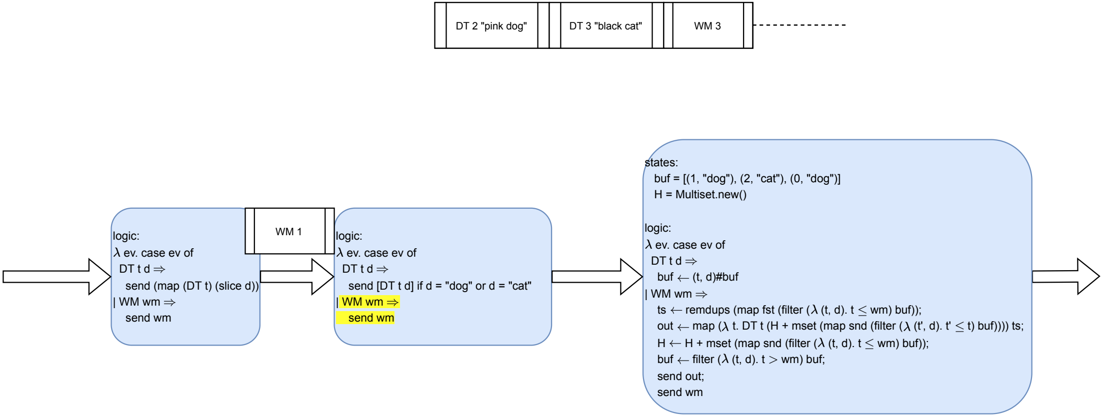
Time-Aware Stream Processing Example



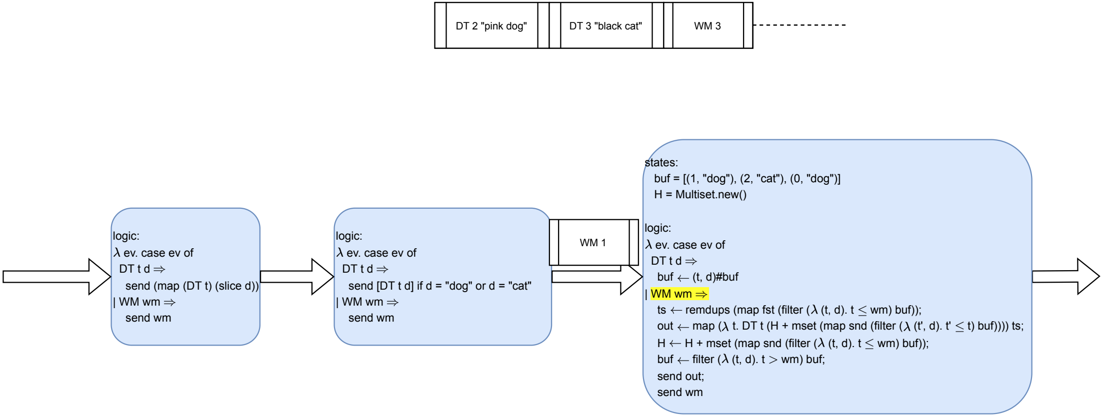
Time-Aware Stream Processing Example



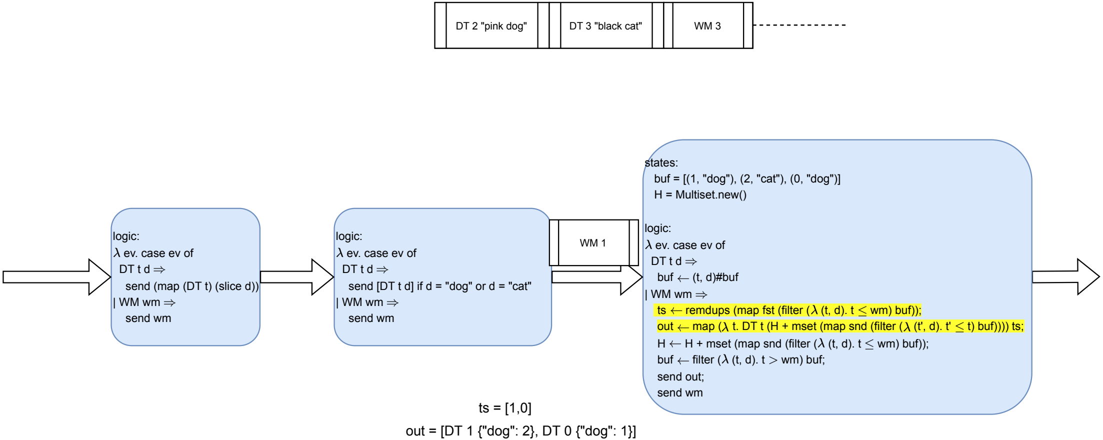
Time-Aware Stream Processing Example



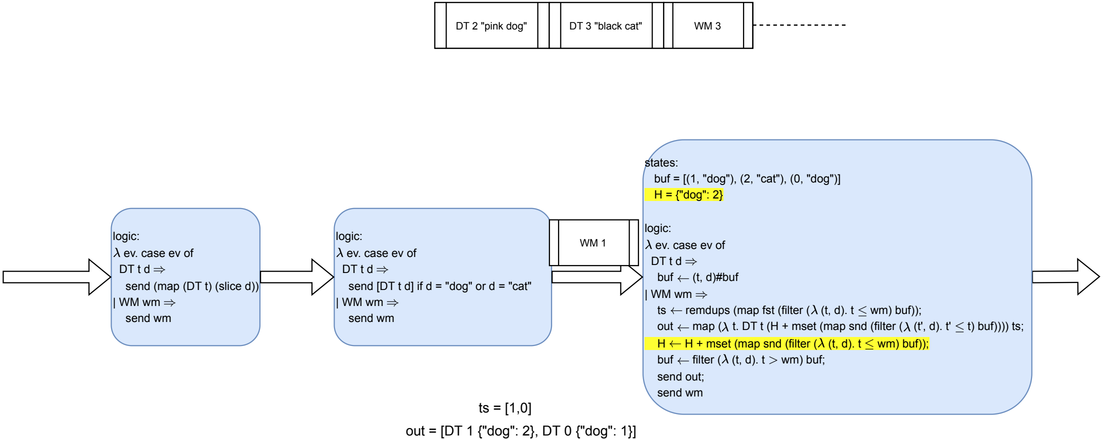
Time-Aware Stream Processing Example



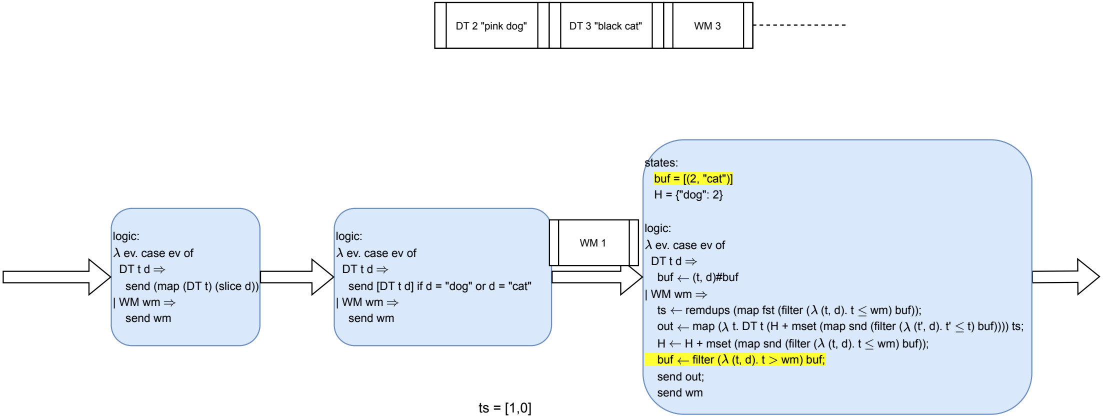
Time-Aware Stream Processing Example



Time-Aware Stream Processing Example



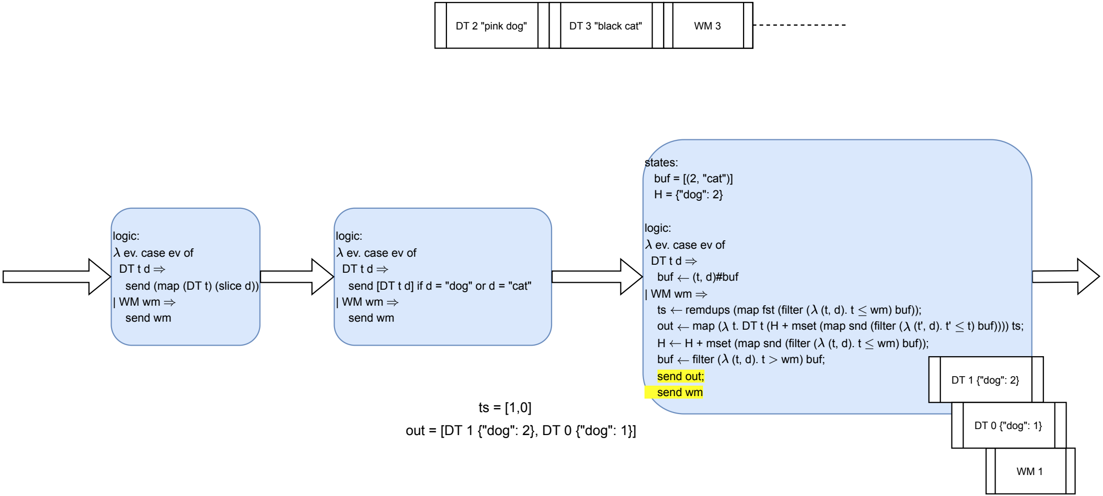
Time-Aware Stream Processing Example



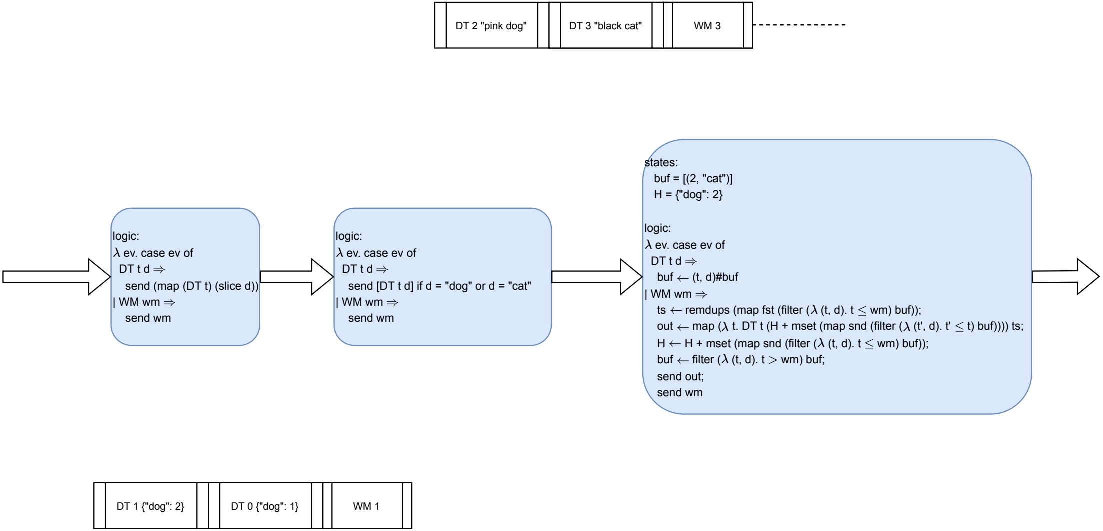
ts = [1,0]

out = [DT 1 {"dog": 2}, DT 0 {"dog": 1}]

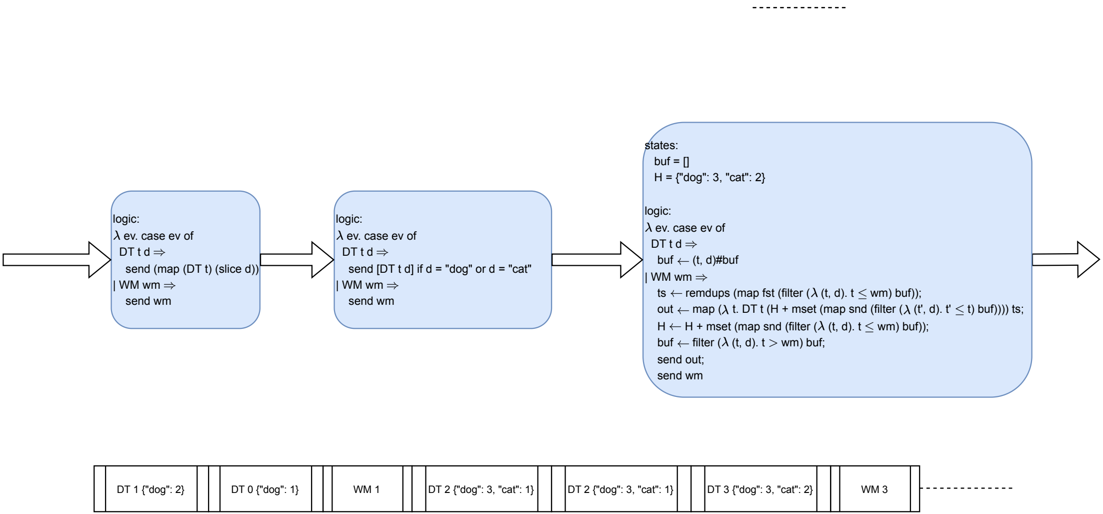
Time-Aware Stream Processing Example



Time-Aware Stream Processing Example



Time-Aware Stream Processing Example



Properties

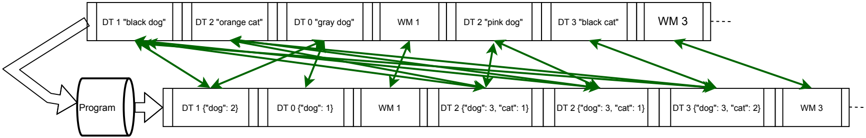
- How do we know if our Dataflow program is what we want?

Properties

- How do we know if our Dataflow program is what we want?
- We need a correctness specification

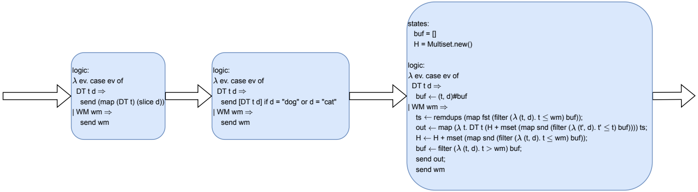
Properties

- How do we know if our Dataflow program is what we want?
- We need a correctness specification
- **Intuition of the specification**:
  - Soundness: for every output $DT\ t\ H$, the "dog" count in H is the count of events with timestamp $(\leq)t$ which contains the string "dog"; similarly for "cat". The count for any other word is always 0.
  - Completeness: The other way around.

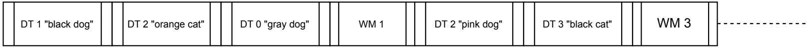


How to prove it

- **lets break down the problem!**:
  - The correctness of the entire Dataflow emerges from the correctness of each part (operator)
    - Operator 1: Slicer
    - Operator 2: Filter
    - Operator 3: Incremental histogram
    - Assumptions about the incoming stream:
    1. Monotone: after WM wm no *DT t d* such that $t \leq wm$.
    2. Productive: after *DT t d* eventually WM wm such that $t \leq wm$



- The original incoming stream must respect monotonicity and productivity



- Each operator must preserve monotonicity and productivity!

Formalization in Isabelle/HOL

Isabelle/HOL: Codatatypes

- Codatatypes

  codatatype *llist* = (lnull: LNil) | LCons (lhd: *'a*) (ltl: *'a llist*)

- Selectors: *lhd*, *ltl*, Discriminator: *lnull*
- Examples:
  - LNil
  - LCons *1* (LCons *2* (LCons *3* LNil))
  - LCons *0* (LCons *0* (LCons *0* (...)))

Isabelle/HOL: Codatatypes

- Codatatypes

  codatatype *llist* = (lnull: LNil) | LCons (lhd: *'a*) (ltl: *'a llist*)

- Selectors: *lhd*, *ltl*, Discriminator: *lnull*
- Examples:
  - LNil
  - LCons *1* (LCons *2* (LCons *3* LNil))
  - LCons *0* (LCons *0* (LCons *0* (. . .)))
- Coinduction principle for lazy list equality

$$
\begin{aligned}
&R\ lxs\ lys \implies \\
&(\bigwedge lxs\ lys.\ R\ lxs\ lys \implies \\
&\quad \text{lnull}\ lxs = \text{lnull}\ lys\ \land \\
&\quad (\neg\ \text{lnull}\ lxs \longrightarrow \neg\ \text{lnull}\ lys \longrightarrow \text{lhd}\ lxs = \text{lhd}\ lys \land R\ (\text{ltl}\ lxs)\ (\text{ltl}\ lys))) \implies \\
&lxs = lys
\end{aligned}
$$

- More about *llist* at the **Coinductive** AFP entry!

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 $b\ c\ s = \ldots$

  - From HOL-Library
  - *None* (Diverged), *Some* x (Finished with final state x)
  - 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

- Recursion: always eventually reduces an argument
- Corecursion: always eventually produces something (productivity)

Isabelle/HOL: Corecursion

- Recursion: always eventually reduces an argument
- Corecursion: always eventually produces something (productivity)
- Corec:

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

$\neg\ \mathsf{lfinite}\ \mathit{lxs} \implies \mathsf{lapp}\ \mathit{lxs}\ \mathit{lys} = \mathit{lxs}$

Isabelle/HOL: (Co)inductive Predicates

- Inductive predicate
  - Finite number of introduction rule applications

$\textbf{inductive}$ in_llist $:: {}'a \Rightarrow {}'a\ llist \Rightarrow bool$ $\textbf{where}$
  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 $(\ldots)$)))

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 (...))))

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 (a.k.a. Non-Time-Aware Stream Processing)

Operator formalization

- Operator as a codatatype[1]
  - Taking $'i$ as the input type, and $'o$ as the output type:
    $\textbf{codatatype}\ ('o, 'i)\ op = \mathrm{Logic}\ (\mathrm{apply}\!: ('i \Rightarrow ('o, 'i)\ op \times 'o\ list))$

[1] This is a simplification of the codatatype in the paper!

Operator formalization

- Operator as a codatatype[1]
  - Taking $'i$ as the input type, and $'o$ as the output type:

    codatatype $('o, 'i)$ *op* = Logic (apply: $('i \Rightarrow ('o, 'i)\ op \times 'o\ list)$)
  - Infinite trees: applying the selector apply "walks" a branch of the tree

corec count_op **where** count_op $P\ n$ =
  Logic ($\lambda e.$ `if` $P\ e$ `then` (count_op $P\ (n + 1)$, $[n{+}1]$) `else` (count_op $P\ n$, $[]$))

apply (count_op *is_even* 0) 0 = (count_op *is_even* 1, $[1]$)

[1] This is a simplification of the codatatype in the paper!

Execution formalization

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

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

$\textbf{corec}$ produce $\textbf{where}$
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'))

- lshift (@@) is a friend of corec function![2]

[2] More about it at the "Friends with benefits" paper by Jasmin Christian Blanchette, Aymeric Bouzy, Andreas Lochbihler, Andrei Popescu, and Dmitriy Traytel

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'*))

- lshift (@@) is a friend of corec function![2]
- produce$_1$ has an induction principle based on the while invariant rule

[2] More about it at the "Friends with benefits" paper by Jasmin Christian Blanchette, Aymeric Bouzy, Andreas Lochbihler, Andrei Popescu, and Dmitriy Traytel

Operators: Count

- Example:

corec count_op **where** count_op P n =
  Logic ($\lambda e.$ `if` P e `then` (count_op P $(n + 1)$, $[n{+}1]$) `else` (count_op P n, $[]$))

$$stream_1 = \boxed{0}\,\boxed{3}\,\boxed{3}\,\boxed{6}\,\boxed{24}$$

$$\text{produce (count\_op } \textit{is\_even } 0) \; stream_1 = \boxed{1}\,\boxed{2}\,\boxed{3}$$

Sequential Composition operator

- Sequential composition: take the output of the first operator and give it as input to the second operator.
- Correctness[3]:

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

- Proof: coinduction principle for lazy list equality and produce_1 induction principle

[3] This is a simplification of the lemma in the paper!

Time-Aware Operators

Time-Aware Streams

- Time-Aware lazy lists

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

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
    - Timely Dataflow: uses pairs to count data cycles (e.g. *(data round n, at cycle i)*)
  - Operators with multiple inputs
    - sum type: represents the multiple input/output ports
- Productive and monotone streams: Coinductive predicates over lazy lists of events.

Proving histogram correct: building Blocks

- Histogram operator: batching and incremental computing
- Building Blocks: reusable operators
  - Batching: batch_op
  - Incremental computing: incr_op
  - Soundness, completeness, preservation of monotonicity and productivity
- Define histogram using the building blocks
- Compositional Reasoning: correctness follows from the correctness of the building blocks
  - Building blocks: more than 9000 loc
  - Incremental Histogram: 300 loc

Batching Operator

The *batch_op* operator[4]:

$$\textbf{fun}\ \mathit{batches}\ \textbf{where}\ \mathit{batches}\ []\ \mathit{tds} = ([], \mathit{tds})$$
$$|\ \ \mathit{batches}\ (\mathit{wm}\ \#\ \mathrm{wms})\ \mathit{tds} = \mathtt{let}\ (\mathit{bs}, \mathit{tds}') = \mathit{batches}\ \mathrm{wms}\ [(t, d) \leftarrow \mathit{tds}.\ \neg\ t \le \mathit{wm}]$$
$$\mathtt{in}\ (\mathrm{DT}\ \mathit{wm}\ [(t, d) \leftarrow \mathit{tds}.\ t \le \mathit{wm}]\ \#\ \mathit{bs}, \mathit{tds}')$$

$$\textbf{corec}\ \mathrm{batch\_op}\ \textbf{where}$$
$$\mathrm{batch\_op}\ \mathit{tds} = \mathrm{Logic}\ (\lambda \mathit{ev}.\ \mathtt{case}\ \mathit{ev}\ \mathtt{of}\ \mathrm{DT}\ t\ d \Rightarrow (\mathrm{batch\_op}\ (\mathit{tds}\ @\ [(t, d)]), [])$$
$$|\ \mathrm{WM}\ \mathit{wm} \Rightarrow \mathtt{let}\ (\mathit{out}, \mathit{tds}') = \mathit{batches}\ [\mathit{wm}]\ \mathit{tds}$$
$$\mathtt{in}\ (\mathrm{batch\_op}\ \mathit{tds}', [x \leftarrow \mathit{out}.\ \mathrm{data}\ x \ne []]\ @\ [\mathrm{WM}\ \mathit{wm}]))$$

$$\mathit{mono}\ \mathit{lxs}\ W \longrightarrow \mathrm{DT}\ \mathit{wm}\ b \in \mathrm{lset}\ (\mathrm{produce}\ (\mathrm{batch\_op}\ \mathit{tds})\ \mathit{lxs}) \longrightarrow$$
$$(\forall t \in \mathrm{set\_t}\ b.\ \mathrm{coll}\ b\ t = \mathrm{lcoll}\ \mathit{lxs}\ t + \mathrm{coll}\ \mathit{tds}\ t) \wedge$$
$$\mathrm{set\_t}\ b = \mathrm{batch\_ts}\ ((\mathrm{map}\ (\lambda\ (t,d).\ \mathrm{DT}\ t\ d)\ \mathit{tds})\ @@\ \mathit{lxs})\ \mathit{wm}$$

$$\mathit{mono}\ \mathit{lxs}\ W \longrightarrow (\forall t \in \mathrm{set\_t}\ \mathit{tds}.\ \forall \mathit{wm} \in W.\ \neg\ t \le \mathit{wm}) \longrightarrow \mathit{mono}\ (\mathrm{produce}\ (\mathrm{batch\_op}\ \mathit{tds})\ \mathit{lxs})\ W$$

[4] Another simplification from the paper

Conclusion

Conclusion

- Isabelle/HOL has a good tool set to formalize and reason about stream processing:
    - Codatatypes, coinductive predicates, corecursion with friends, reasoning up to friends (congruence)
        - Coinduction up to congruence principle is automatically derived for codatatypes (but not for coinductive principles)
- Next step: Feedback loop

Questions, comments and suggestions