Time-Aware Stream Processing in Isabelle/HOL

Rafael Castro G. Silva and Dmitriy Traytel

rasi@di.ku.dk

Department of Computer Science
University of Copenhagen

14/09/2024

Introduction

Time-Aware Stream Processing

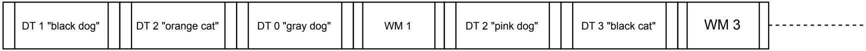
- What is Time-Aware Stream Processing?

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

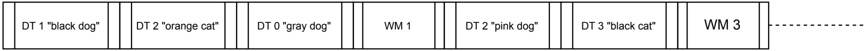
- What is Time-Aware Stream Processing?
  - Stream Processing: programs that compute (possibly) unbounded sequences of data (streams)
  - Time-Aware: Data has timestamp metadata; timestamps are bounded by watermarks
    - Timestamp: A partially-ordered value associated with the data (e.g., unix-time,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



DT 1 "black dog"
DT 2 "orange cat"
DT 0 "gray dog"
WM 1
DT 2 "pink dog"
DT 3 "black cat"
WM 3

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.



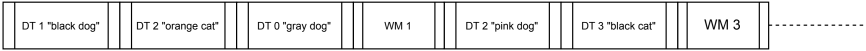
DT 1 "black dog" | DT 2 "orange cat" | DT 0 "gray dog" | WM 1 | DT 2 "pink dog" | DT 3 "black cat" | WM 3

- 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

Cloud DataFlow Flink

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.



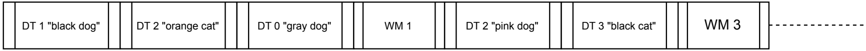
DT 1 "black dog" | DT 2 "orange cat" | DT 0 "gray dog" | WM 1 | DT 2 "pink dog" | DT 3 "black cat" | WM 3

- 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)

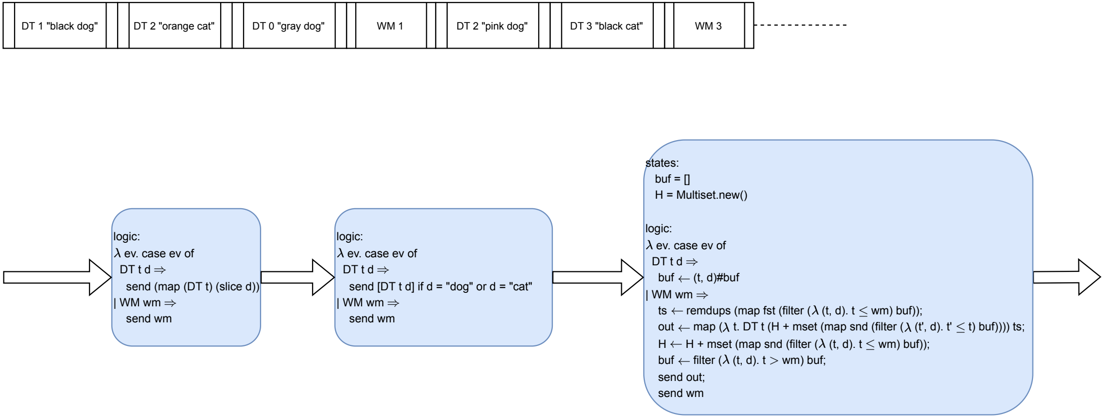
Time-Aware Stream Processing Example

Example:
Incrementally count the occurrences of the words "dog" and "cat"



DT 1 "black dog" | DT 2 "orange cat" | DT 0 "gray dog" | WM 1 | DT 2 "pink dog" | DT 3 "black cat" | WM 3

Time-Aware Stream Processing Example



DT 1 "black dog" | DT 2 "orange cat" | DT 0 "gray dog" | WM 1 | DT 2 "pink dog" | DT 3 "black cat" | WM 3

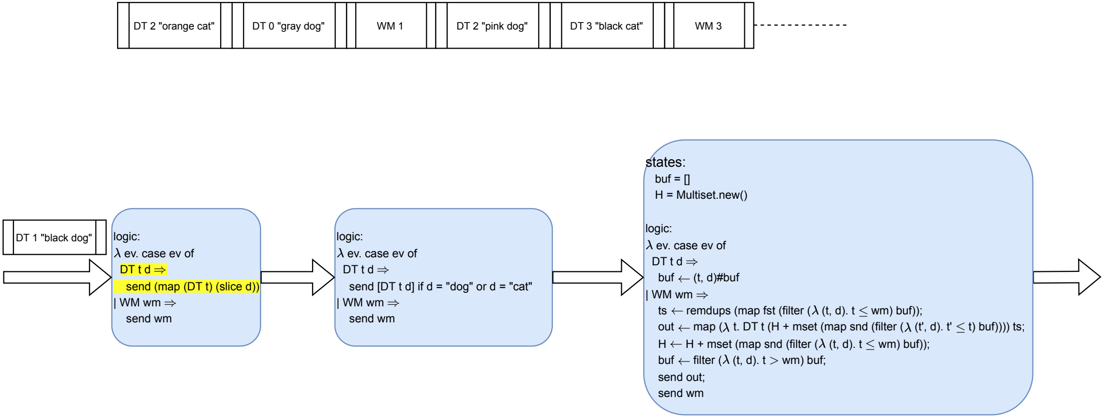
```
logic:
λ ev. case ev of
  DT t d ⇒
    send (map (DT t) (slice d))
| WM wm ⇒
    send wm
```

```
logic:
λ ev. case ev of
  DT t d ⇒
    send [DT t d] if d = "dog" or d = "cat"
| WM wm ⇒
    send wm
```

```
states:
  buf = []
  H = Multiset.new()

logic:
λ ev. case ev of
  DT t d ⇒
    buf ← (t, d)#buf
| WM wm ⇒
    ts ← remdups (map fst (filter (λ (t, d). t ≤ wm) buf));
    out ← map (λ t. DT t (H + mset (map snd (filter (λ (t', d). t' ≤ t) buf)))) ts;
    H ← H + mset (map snd (filter (λ (t, d). t ≤ wm) buf));
    buf ← filter (λ (t, d). t > wm) buf;
    send out;
    send wm
```

Time-Aware Stream Processing Example



DT 2 "orange cat" | DT 0 "gray dog" | WM 1 | DT 2 "pink dog" | DT 3 "black cat" | WM 3

DT 1 "black dog"

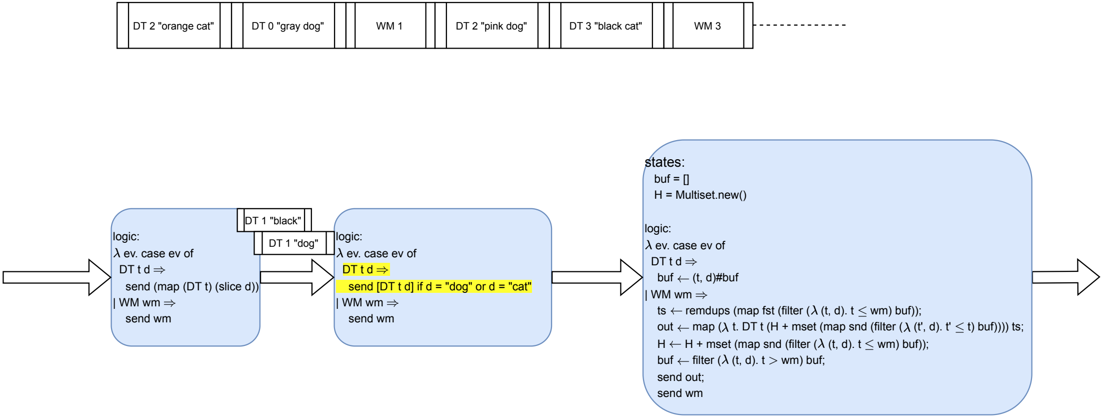
```
logic:
λ ev. case ev of
 DT t d ⇒
   send (map (DT t) (slice d))
| WM wm ⇒
   send wm
```

```
logic:
λ ev. case ev of
 DT t d ⇒
   send [DT t d] if d = "dog" or d = "cat"
| WM wm ⇒
   send wm
```

```
states:
  buf = []
  H = Multiset.new()

logic:
λ ev. case ev of
 DT t d ⇒
   buf ← (t, d)#buf
| WM wm ⇒
   ts ← remdups (map fst (filter (λ (t, d). t ≤ wm) buf));
   out ← map (λ t. DT t (H + mset (map snd (filter (λ (t', d). t' ≤ t) buf)))) ts;
   H ← H + mset (map snd (filter (λ (t, d). t ≤ wm) buf));
   buf ← filter (λ (t, d). t > wm) buf;
   send out;
   send wm
```

Time-Aware Stream Processing Example



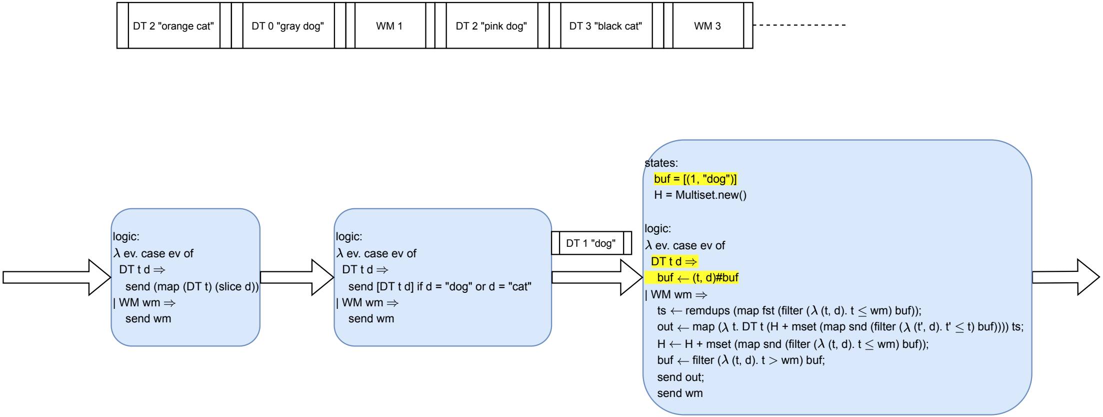
```
logic:
λ ev. case ev of
 DT t d ⇒
   send (map (DT t) (slice d))
| WM wm ⇒
   send wm
```

```
logic:
λ ev. case ev of
 DT t d ⇒
   send [DT t d] if d = "dog" or d = "cat"
| WM wm ⇒
   send wm
```

```
states:
  buf = []
  H = Multiset.new()

logic:
λ ev. case ev of
  DT t d ⇒
    buf ← (t, d)#buf
| WM wm ⇒
    ts ← remdups (map fst (filter (λ (t, d). t ≤ wm) buf));
    out ← map (λ t. DT t (H + mset (map snd (filter (λ (t', d). t' ≤ t) buf)))) ts;
    H ← H + mset (map snd (filter (λ (t, d). t ≤ wm) buf));
    buf ← filter (λ (t, d). t > wm) buf;
    send out;
    send wm
```

Time-Aware Stream Processing Example



DT 2 "orange cat" | DT 0 "gray dog" | WM 1 | DT 2 "pink dog" | DT 3 "black cat" | WM 3 | - - - - - - -

```
logic:
λ ev. case ev of
 DT t d ⇒
   send (map (DT t) (slice d))
| WM wm ⇒
   send wm
```

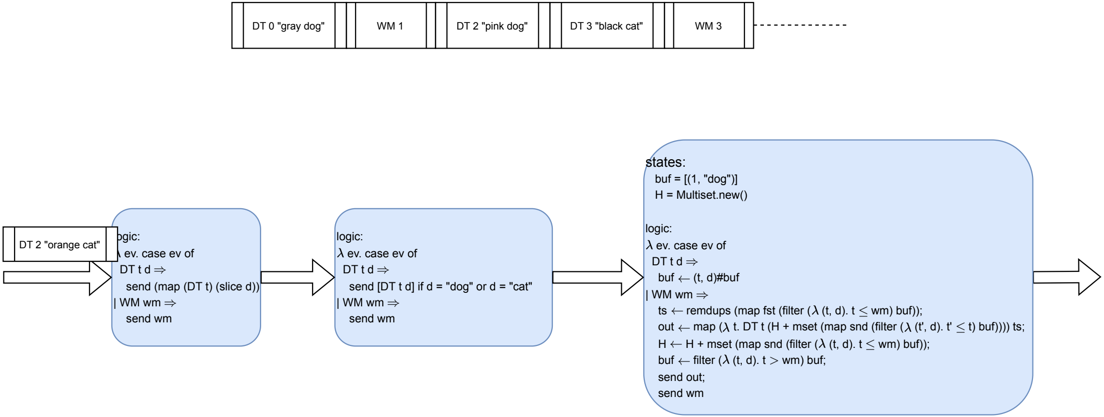
```
logic:
λ ev. case ev of
 DT t d ⇒
   send [DT t d] if d = "dog" or d = "cat"
| WM wm ⇒
   send wm
```

DT 1 "dog"

```
states:
  buf = [(1, "dog")]
  H = Multiset.new()

logic:
λ ev. case ev of
  DT t d ⇒
    buf ← (t, d)#buf
| WM wm ⇒
    ts ← remdups (map fst (filter (λ (t, d). t ≤ wm) buf));
    out ← map (λ t. DT t (H + mset (map snd (filter (λ (t', d). t' ≤ t) buf)))) ts;
    H ← H + mset (map snd (filter (λ (t, d). t ≤ wm) buf));
    buf ← filter (λ (t, d). t > wm) buf;
    send out;
    send wm
```

Time-Aware Stream Processing Example



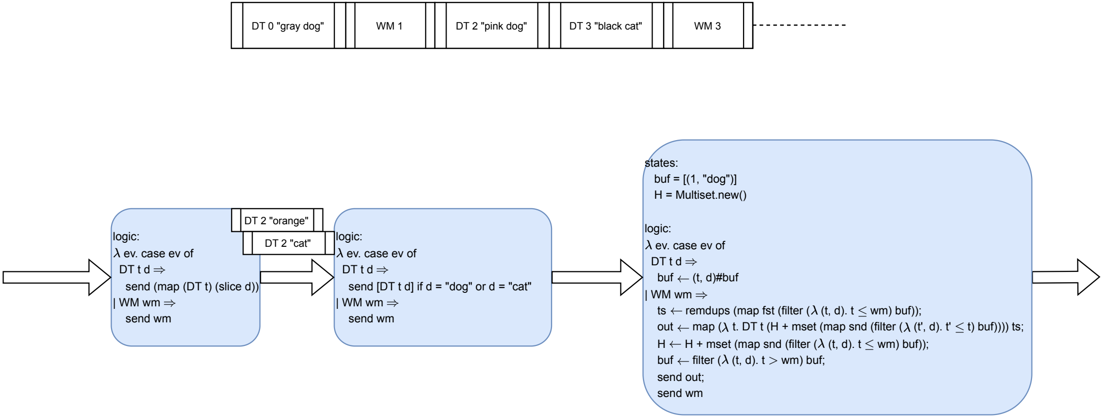
```
ogic:
 ev. case ev of
  DT t d ⇒
    send (map (DT t) (slice d))
| WM wm ⇒
    send wm
```

```
logic:
λ ev. case ev of
  DT t d ⇒
    send [DT t d] if d = "dog" or d = "cat"
| WM wm ⇒
    send wm
```

```
states:
  buf = [(1, "dog")]
  H = Multiset.new()

logic:
λ ev. case ev of
  DT t d ⇒
    buf ← (t, d)#buf
| WM wm ⇒
    ts ← remdups (map fst (filter (λ (t, d). t ≤ wm) buf));
    out ← map (λ t. DT t (H + mset (map snd (filter (λ (t', d). t' ≤ t) buf)))) ts;
    H ← H + mset (map snd (filter (λ (t, d). t ≤ wm) buf));
    buf ← filter (λ (t, d). t > wm) buf;
    send out;
    send wm
```

Time-Aware Stream Processing Example



DT 0 "gray dog" | WM 1 | DT 2 "pink dog" | DT 3 "black cat" | WM 3

DT 2 "orange"

DT 2 "cat"

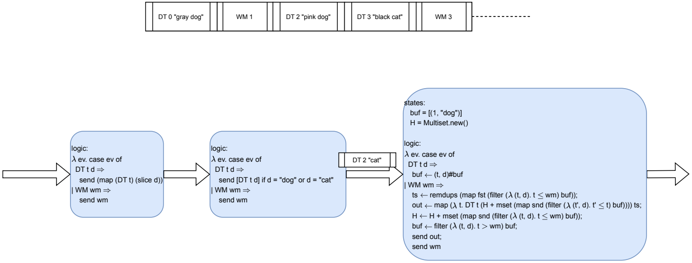
```
logic:
λ ev. case ev of
  DT t d ⇒
    send (map (DT t) (slice d))
| WM wm ⇒
    send wm
```

```
logic:
λ ev. case ev of
  DT t d ⇒
    send [DT t d] if d = "dog" or d = "cat"
| WM wm ⇒
    send wm
```

```
states:
  buf = [(1, "dog")]
  H = Multiset.new()

logic:
λ ev. case ev of
  DT t d ⇒
    buf ← (t, d)#buf
| WM wm ⇒
    ts ← remdups (map fst (filter (λ (t, d). t ≤ wm) buf));
    out ← map (λ t. DT t (H + mset (map snd (filter (λ (t', d). t' ≤ t) buf)))) ts;
    H ← H + mset (map snd (filter (λ (t, d). t ≤ wm) buf));
    buf ← filter (λ (t, d). t > wm) buf;
    send out;
    send wm
```

Time-Aware Stream Processing Example



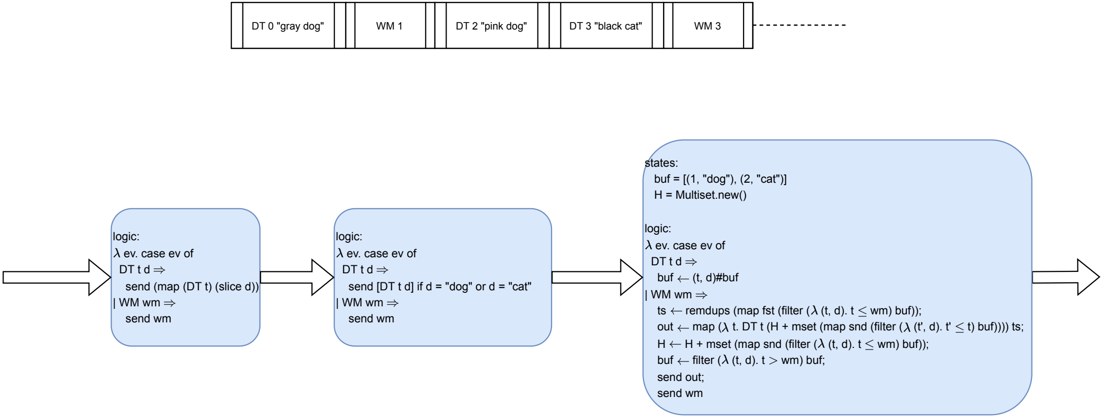
```
logic:
λ ev. case ev of
  DT t d ⇒
    send (map (DT t) (slice d))
| WM wm ⇒
    send wm
```

```
logic:
λ ev. case ev of
  DT t d ⇒
    send [DT t d] if d = "dog" or d = "cat"
| WM wm ⇒
    send wm
```

```
states:
   buf = [(1, "dog")]
   H = Multiset.new()

logic:
λ ev. case ev of
  DT t d ⇒
    buf ← (t, d)#buf
| WM wm ⇒
    ts ← remdups (map fst (filter (λ (t, d). t ≤ wm) buf));
    out ← map (λ t. DT t (H + mset (map snd (filter (λ (t', d). t' ≤ t) buf)))) ts;
    H ← H + mset (map snd (filter (λ (t, d). t ≤ wm) buf));
    buf ← filter (λ (t, d). t > wm) buf;
    send out;
    send wm
```

Time-Aware Stream Processing Example



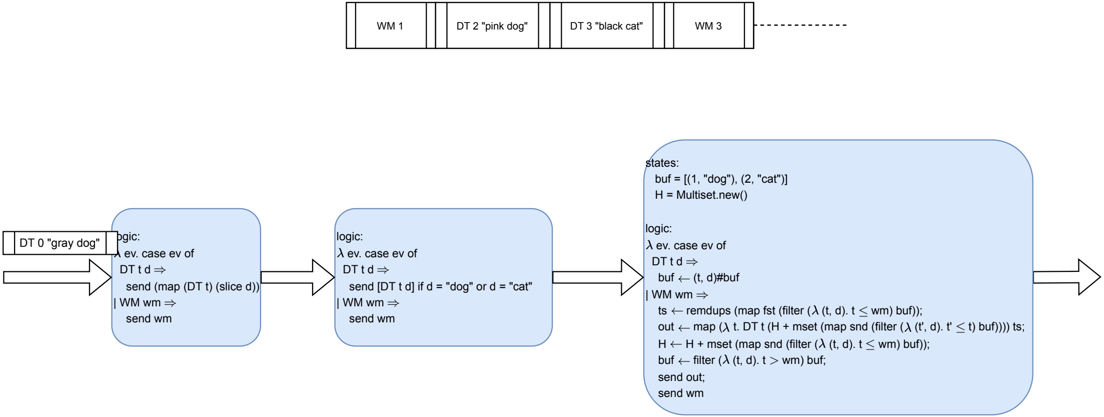
```
logic:
λ ev. case ev of
  DT t d ⇒
    send (map (DT t) (slice d))
| WM wm ⇒
    send wm
```

```
logic:
λ ev. case ev of
  DT t d ⇒
    send [DT t d] if d = "dog" or d = "cat"
| WM wm ⇒
    send wm
```

```
states:
  buf = [(1, "dog"), (2, "cat")]
  H = Multiset.new()

logic:
λ ev. case ev of
  DT t d ⇒
    buf ← (t, d)#buf
| WM wm ⇒
    ts ← remdups (map fst (filter (λ (t, d). t ≤ wm) buf));
    out ← map (λ t. DT t (H + mset (map snd (filter (λ (t', d). t' ≤ t) buf)))) ts;
    H ← H + mset (map snd (filter (λ (t, d). t ≤ wm) buf));
    buf ← filter (λ (t, d). t > wm) buf;
    send out;
    send wm
```

Time-Aware Stream Processing Example



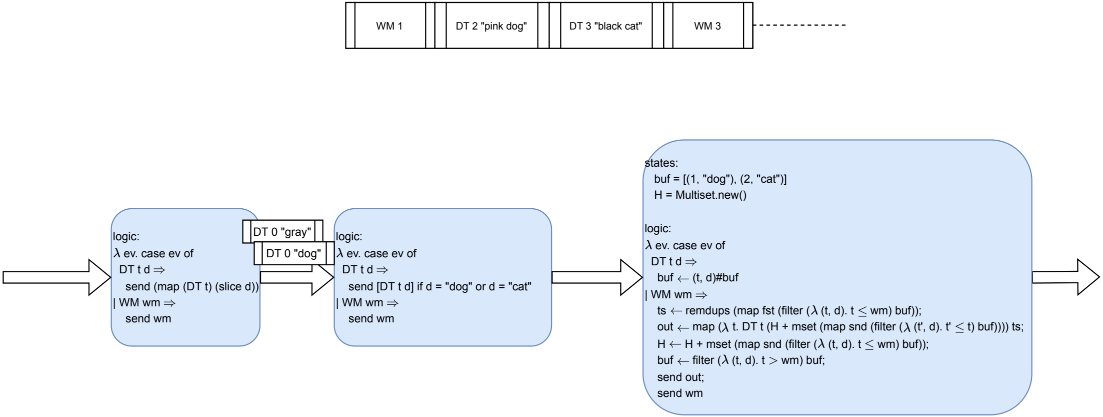
```
logic:
λ ev. case ev of
 DT t d ⇒
   send (map (DT t) (slice d))
| WM wm ⇒
   send wm
```

```
logic:
λ ev. case ev of
 DT t d ⇒
   send [DT t d] if d = "dog" or d = "cat"
| WM wm ⇒
   send wm
```

```
states:
  buf = [(1, "dog"), (2, "cat")]
  H = Multiset.new()

logic:
λ ev. case ev of
  DT t d ⇒
    buf ← (t, d)#buf
| WM wm ⇒
    ts ← remdups (map fst (filter (λ (t, d). t ≤ wm) buf));
    out ← map (λ t. DT t (H + mset (map snd (filter (λ (t', d). t' ≤ t) buf)))) ts;
    H ← H + mset (map snd (filter (λ (t, d). t ≤ wm) buf));
    buf ← filter (λ (t, d). t > wm) buf;
    send out;
    send wm
```

Time-Aware Stream Processing Example



WM 1 | DT 2 "pink dog" | DT 3 "black cat" | WM 3

DT 0 "gray"

DT 0 "dog"

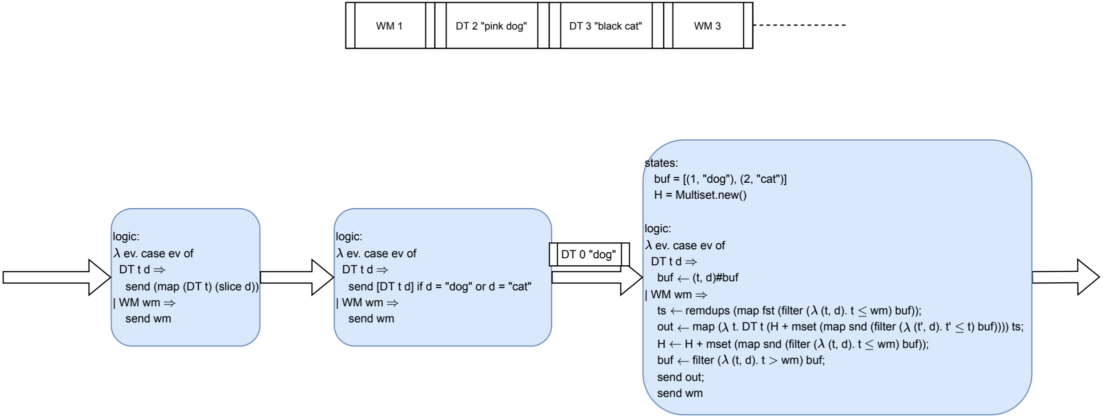
```
logic:
λ ev. case ev of
 DT t d ⇒
  send (map (DT t) (slice d))
| WM wm ⇒
  send wm
```

```
logic:
λ ev. case ev of
 DT t d ⇒
  send [DT t d] if d = "dog" or d = "cat"
| WM wm ⇒
  send wm
```

```
states:
  buf = [(1, "dog"), (2, "cat")]
  H = Multiset.new()

logic:
λ ev. case ev of
 DT t d ⇒
  buf ← (t, d)#buf
| WM wm ⇒
  ts ← remdups (map fst (filter (λ (t, d). t ≤ wm) buf));
  out ← map (λ t. DT t (H + mset (map snd (filter (λ (t', d). t' ≤ t) buf)))) ts;
  H ← H + mset (map snd (filter (λ (t, d). t ≤ wm) buf));
  buf ← filter (λ (t, d). t > wm) buf;
  send out;
  send wm
```

Time-Aware Stream Processing Example



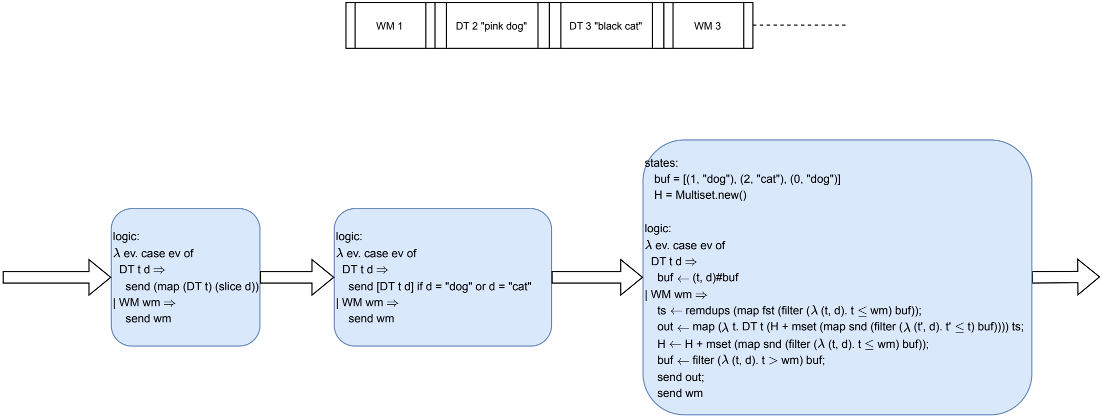
```
logic:
λ ev. case ev of
  DT t d ⇒
    send (map (DT t) (slice d))
| WM wm ⇒
    send wm
```

```
logic:
λ ev. case ev of
  DT t d ⇒
    send [DT t d] if d = "dog" or d = "cat"
| WM wm ⇒
    send wm
```

```
states:
  buf = [(1, "dog"), (2, "cat")]
  H = Multiset.new()

logic:
λ ev. case ev of
  DT t d ⇒
    buf ← (t, d)#buf
| WM wm ⇒
    ts ← remdups (map fst (filter (λ (t, d). t ≤ wm) buf));
    out ← map (λ t. DT t (H + mset (map snd (filter (λ (t', d). t' ≤ t) buf)))) ts;
    H ← H + mset (map snd (filter (λ (t, d). t ≤ wm) buf));
    buf ← filter (λ (t, d). t > wm) buf;
    send out;
    send wm
```

Time-Aware Stream Processing Example



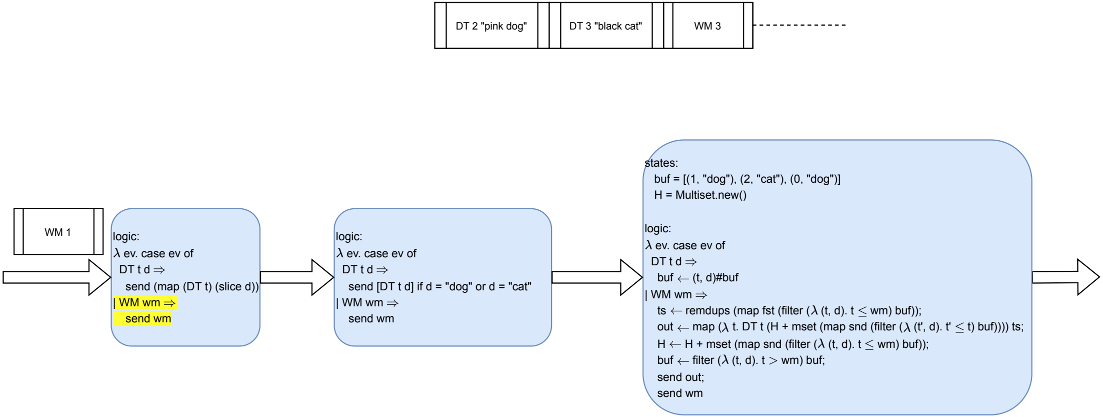
```
logic:
λ ev. case ev of
  DT t d ⇒
    send (map (DT t) (slice d))
| WM wm ⇒
    send wm
```

```
logic:
λ ev. case ev of
  DT t d ⇒
    send [DT t d] if d = "dog" or d = "cat"
| WM wm ⇒
    send wm
```

```
states:
  buf = [(1, "dog"), (2, "cat"), (0, "dog")]
  H = Multiset.new()

logic:
λ ev. case ev of
  DT t d ⇒
    buf ← (t, d)#buf
| WM wm ⇒
    ts ← remdups (map fst (filter (λ (t, d). t ≤ wm) buf));
    out ← map (λ t. DT t (H + mset (map snd (filter (λ (t', d). t' ≤ t) buf)))) ts;
    H ← H + mset (map snd (filter (λ (t, d). t ≤ wm) buf));
    buf ← filter (λ (t, d). t > wm) buf;
    send out;
    send wm
```

Time-Aware Stream Processing Example



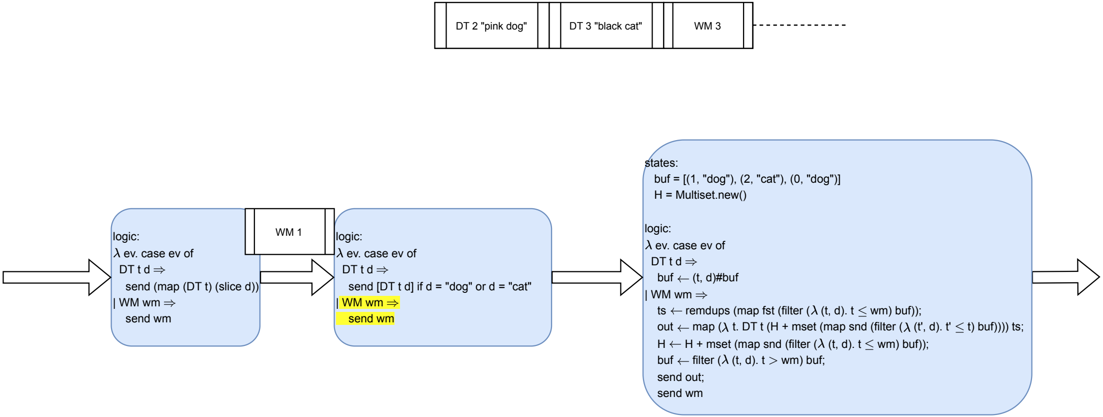
```
logic:
λ ev. case ev of
  DT t d ⇒
    send (map (DT t) (slice d))
| WM wm ⇒
    send wm
```

```
logic:
λ ev. case ev of
  DT t d ⇒
    send [DT t d] if d = "dog" or d = "cat"
| WM wm ⇒
    send wm
```

```
states:
   buf = [(1, "dog"), (2, "cat"), (0, "dog")]
   H = Multiset.new()

logic:
λ ev. case ev of
  DT t d ⇒
    buf ← (t, d)#buf
| WM wm ⇒
    ts ← remdups (map fst (filter (λ (t, d). t ≤ wm) buf));
    out ← map (λ t. DT t (H + mset (map snd (filter (λ (t', d). t' ≤ t) buf)))) ts;
    H ← H + mset (map snd (filter (λ (t, d). t ≤ wm) buf));
    buf ← filter (λ (t, d). t > wm) buf;
    send out;
    send wm
```

Time-Aware Stream Processing Example



DT 2 "pink dog" | DT 3 "black cat" | WM 3

```
logic:
λ ev. case ev of
 DT t d ⇒
   send (map (DT t) (slice d))
| WM wm ⇒
   send wm
```

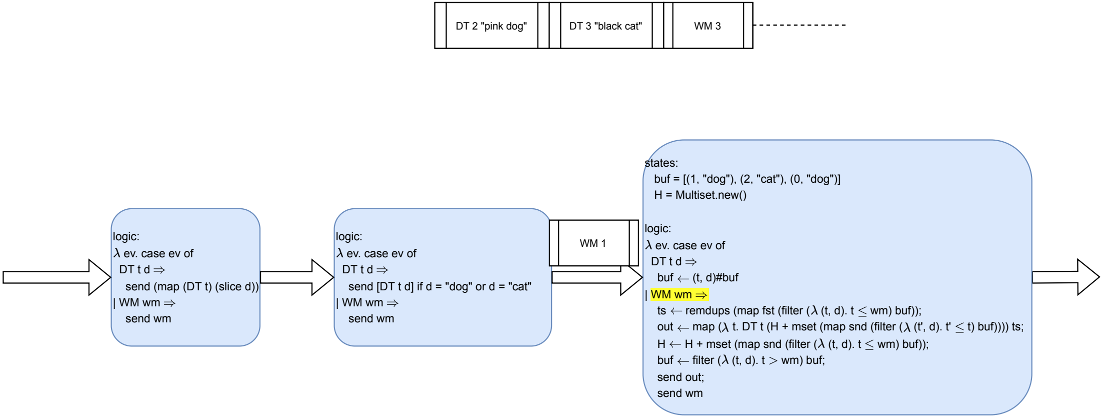
WM 1

```
logic:
λ ev. case ev of
 DT t d ⇒
   send [DT t d] if d = "dog" or d = "cat"
| WM wm ⇒
   send wm
```

```
states:
  buf = [(1, "dog"), (2, "cat"), (0, "dog")]
  H = Multiset.new()

logic:
λ ev. case ev of
  DT t d ⇒
    buf ← (t, d)#buf
| WM wm ⇒
    ts ← remdups (map fst (filter (λ (t, d). t ≤ wm) buf));
    out ← map (λ t. DT t (H + mset (map snd (filter (λ (t', d). t' ≤ t) buf)))) ts;
    H ← H + mset (map snd (filter (λ (t, d). t ≤ wm) buf));
    buf ← filter (λ (t, d). t > wm) buf;
    send out;
    send wm
```

Time-Aware Stream Processing Example



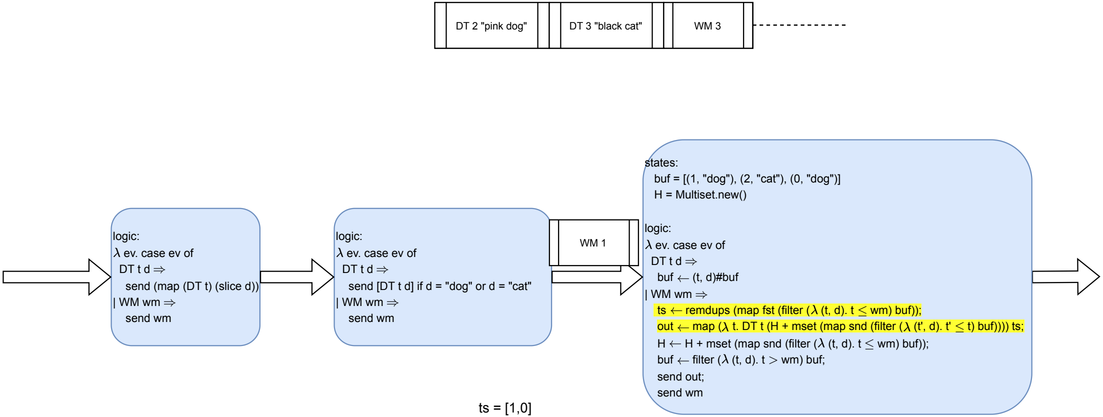
```
logic:
λ ev. case ev of
  DT t d ⇒
    send (map (DT t) (slice d))
| WM wm ⇒
    send wm
```

```
logic:
λ ev. case ev of
  DT t d ⇒
    send [DT t d] if d = "dog" or d = "cat"
| WM wm ⇒
    send wm
```

```
states:
   buf = [(1, "dog"), (2, "cat"), (0, "dog")]
   H = Multiset.new()

logic:
λ ev. case ev of
  DT t d ⇒
    buf ← (t, d)#buf
| WM wm ⇒
    ts ← remdups (map fst (filter (λ (t, d). t ≤ wm) buf));
    out ← map (λ t. DT t (H + mset (map snd (filter (λ (t', d). t' ≤ t) buf)))) ts;
    H ← H + mset (map snd (filter (λ (t, d). t ≤ wm) buf));
    buf ← filter (λ (t, d). t > wm) buf;
    send out;
    send wm
```

Time-Aware Stream Processing Example



```
logic:
λ ev. case ev of
  DT t d ⇒
    send (map (DT t) (slice d))
| WM wm ⇒
    send wm
```

```
logic:
λ ev. case ev of
  DT t d ⇒
    send [DT t d] if d = "dog" or d = "cat"
| WM wm ⇒
    send wm
```

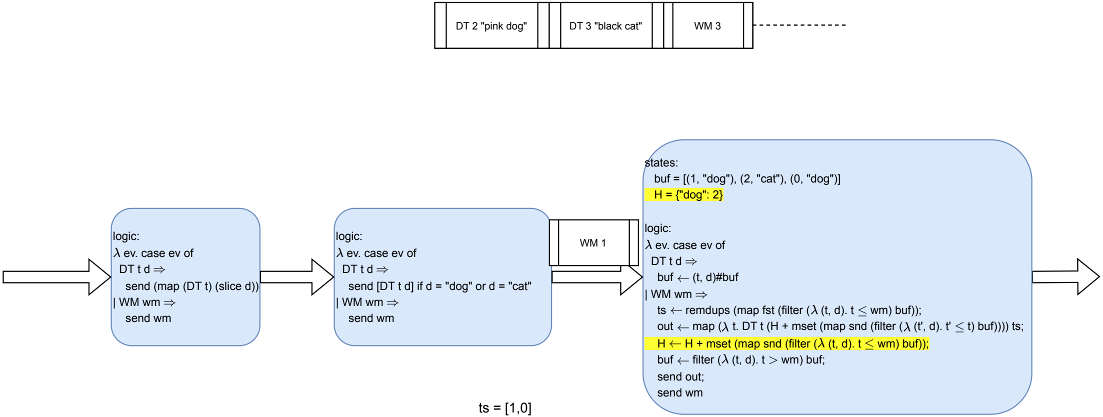
```
states:
  buf = [(1, "dog"), (2, "cat"), (0, "dog")]
  H = Multiset.new()

logic:
λ ev. case ev of
  DT t d ⇒
    buf ← (t, d)#buf
| WM wm ⇒
    ts ← remdups (map fst (filter (λ (t, d). t ≤ wm) buf));
    out ← map (λ t. DT t (H + mset (map snd (filter (λ (t', d). t' ≤ t) buf)))) ts;
    H ← H + mset (map snd (filter (λ (t, d). t ≤ wm) buf));
    buf ← filter (λ (t, d). t > wm) buf;
    send out;
    send wm
```

ts = [1,0]

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

Time-Aware Stream Processing Example



```
logic:
λ ev. case ev of
  DT t d ⇒
    send (map (DT t) (slice d))
| WM wm ⇒
    send wm
```

```
logic:
λ ev. case ev of
  DT t d ⇒
    send [DT t d] if d = "dog" or d = "cat"
| WM wm ⇒
    send wm
```

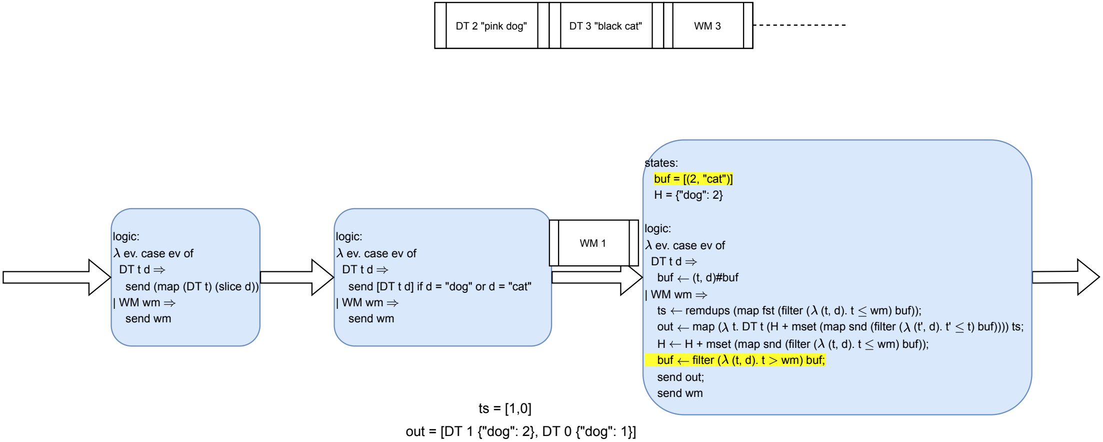
```
states:
  buf = [(1, "dog"), (2, "cat"), (0, "dog")]
  H = {"dog": 2}

logic:
λ ev. case ev of
  DT t d ⇒
    buf ← (t, d)#buf
| WM wm ⇒
    ts ← remdups (map fst (filter (λ (t, d). t ≤ wm) buf));
    out ← map (λ t. DT t (H + mset (map snd (filter (λ (t', d). t' ≤ t) buf)))) ts;
    H ← H + mset (map snd (filter (λ (t, d). t ≤ wm) buf));
    buf ← filter (λ (t, d). t > wm) buf;
    send out;
    send wm
```

ts = [1,0]

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

Time-Aware Stream Processing Example



```
logic:
λ ev. case ev of
  DT t d ⇒
    send (map (DT t) (slice d))
| WM wm ⇒
    send wm
```

```
logic:
λ ev. case ev of
  DT t d ⇒
    send [DT t d] if d = "dog" or d = "cat"
| WM wm ⇒
    send wm
```

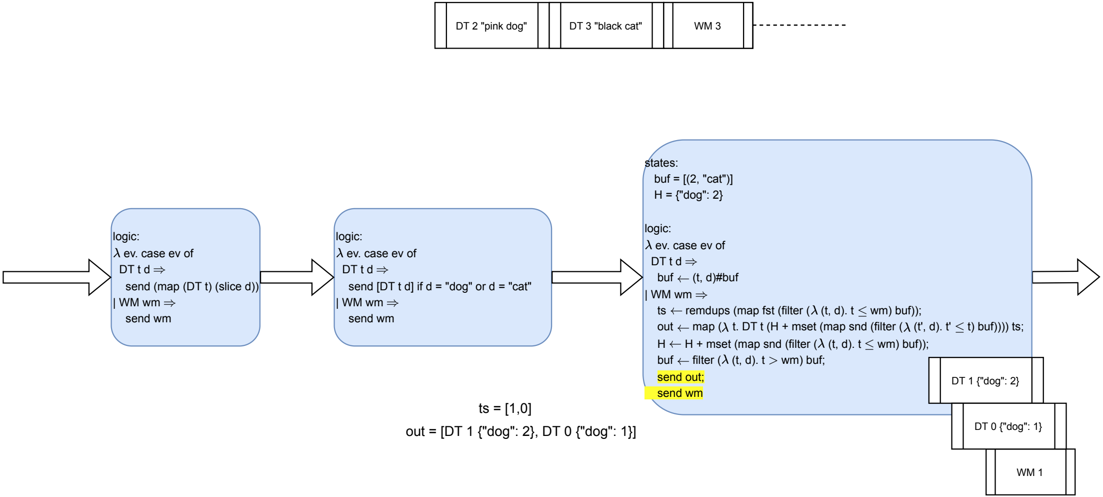
```
states:
  buf = [(2, "cat")]
  H = {"dog": 2}

logic:
λ ev. case ev of
  DT t d ⇒
    buf ← (t, d)#buf
| WM wm ⇒
    ts ← remdups (map fst (filter (λ (t, d). t ≤ wm) buf));
    out ← map (λ t. DT t (H + mset (map snd (filter (λ (t', d). t' ≤ t) buf)))) ts;
    H ← H + mset (map snd (filter (λ (t, d). t ≤ wm) buf));
    buf ← filter (λ (t, d). t > wm) buf;
    send out;
    send wm
```

ts = [1,0]

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

Time-Aware Stream Processing Example



```
logic:
λ ev. case ev of
 DT t d ⇒
   send (map (DT t) (slice d))
| WM wm ⇒
   send wm
```

```
logic:
λ ev. case ev of
 DT t d ⇒
   send [DT t d] if d = "dog" or d = "cat"
| WM wm ⇒
   send wm
```

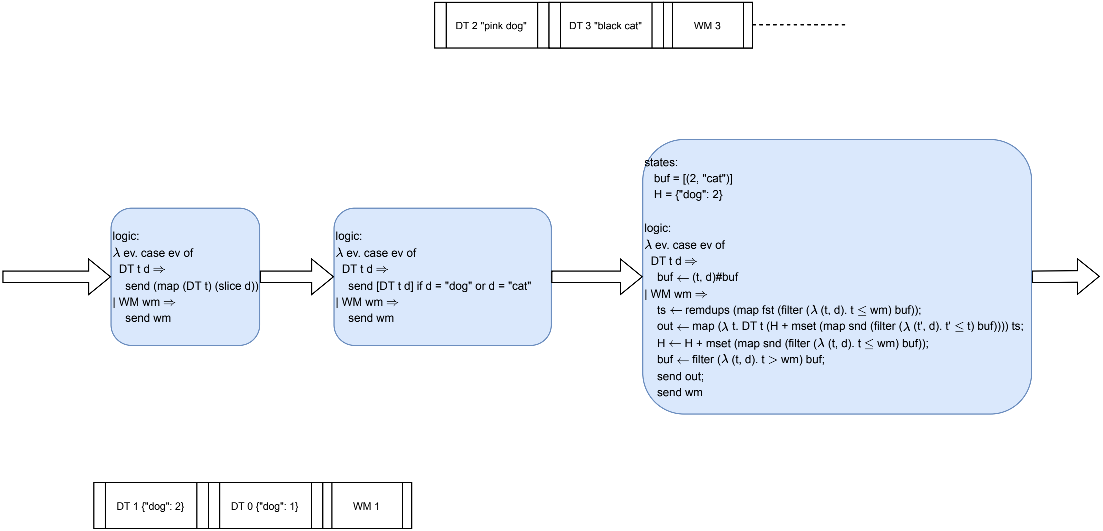
```
states:
  buf = [(2, "cat")]
  H = {"dog": 2}

logic:
λ ev. case ev of
  DT t d ⇒
    buf ← (t, d)#buf
| WM wm ⇒
    ts ← remdups (map fst (filter (λ (t, d). t ≤ wm) buf));
    out ← map (λ t. DT t (H + mset (map snd (filter (λ (t', d). t' ≤ t) buf)))) ts;
    H ← H + mset (map snd (filter (λ (t, d). t ≤ wm) buf));
    buf ← filter (λ (t, d). t > wm) buf;
    send out;
    send wm
```

ts = [1,0]

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

Time-Aware Stream Processing Example



DT 2 "pink dog" | DT 3 "black cat" | WM 3

```
logic:
λ ev. case ev of
  DT t d ⇒
    send (map (DT t) (slice d))
| WM wm ⇒
    send wm
```

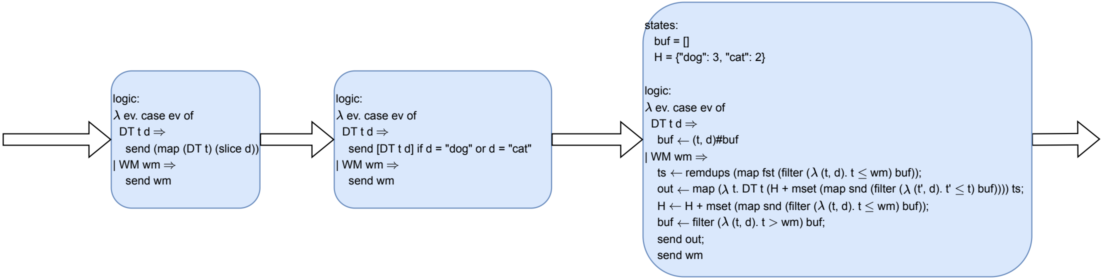
```
logic:
λ ev. case ev of
  DT t d ⇒
    send [DT t d] if d = "dog" or d = "cat"
| WM wm ⇒
    send wm
```

```
states:
  buf = [(2, "cat")]
  H = {"dog": 2}

logic:
λ ev. case ev of
  DT t d ⇒
    buf ← (t, d)#buf
| WM wm ⇒
    ts ← remdups (map fst (filter (λ (t, d). t ≤ wm) buf));
    out ← map (λ t. DT t (H + mset (map snd (filter (λ (t', d). t' ≤ t) buf)))) ts;
    H ← H + mset (map snd (filter (λ (t, d). t ≤ wm) buf));
    buf ← filter (λ (t, d). t > wm) buf;
    send out;
    send wm
```

DT 1 {"dog": 2} | DT 0 {"dog": 1} | WM 1

Time-Aware Stream Processing Example



```
logic:
λ ev. case ev of
  DT t d ⇒
    send (map (DT t) (slice d))
| WM wm ⇒
    send wm
```

```
logic:
λ ev. case ev of
  DT t d ⇒
    send [DT t d] if d = "dog" or d = "cat"
| WM wm ⇒
    send wm
```

```
states:
  buf = []
  H = {"dog": 3, "cat": 2}

logic:
λ ev. case ev of
  DT t d ⇒
    buf ← (t, d)#buf
| WM wm ⇒
    ts ← remdups (map fst (filter (λ (t, d). t ≤ wm) buf));
    out ← map (λ t. DT t (H + mset (map snd (filter (λ (t', d). t' ≤ t) buf)))) ts;
    H ← H + mset (map snd (filter (λ (t, d). t ≤ wm) buf));
    buf ← filter (λ (t, d). t > wm) buf;
    send out;
    send wm
```

DT 1 {"dog": 2}	DT 0 {"dog": 1}	WM 1	DT 2 {"dog": 3, "cat": 1}	DT 2 {"dog": 3, "cat": 1}	DT 3 {"dog": 3, "cat": 2}	WM 3

Properties

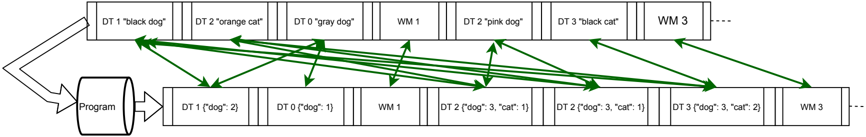
- How do we know if our Dataflow program is what we want?

Properties

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

Properties

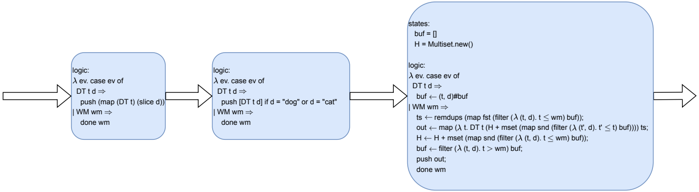
- How do we know if our Dataflow program is what we want?
- We need a correctness specification
- **Intuition of the specification**:
  - Soundness: for every output $DT\ t\ H$, the “dog” count in H is the count of events with timestamp $(\leq) t$ which contains the string “dog”; similarly for “cat”. 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 $t \leq wm$



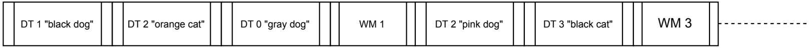
```
logic:
λ ev. case ev of
  DT t d ⇒
    push (map (DT t) (slice d))
| WM wm ⇒
    done wm
```

```
logic:
λ ev. case ev of
  DT t d ⇒
    push [DT t d] if d = "dog" or d = "cat"
| WM wm ⇒
    done wm
```

```
states:
  buf = []
  H = Multiset.new()

logic:
λ ev. case ev of
  DT t d ⇒
    buf ← (t, d)#buf
| WM wm ⇒
    ts ← remdups (map fst (filter (λ (t, d). t ≤ wm) buf));
    out ← map (λ t. DT t (H + mset (map snd (filter (λ (t', d). t' ≤ t) buf)))) ts;
    H ← H + mset (map snd (filter (λ (t, d). t ≤ wm) buf));
    buf ← filter (λ (t, d). t > wm) buf;
    push out;
    done wm
```

- The original incoming stream must respect monotonicity and productivity



DT 1 "black dog" | DT 2 "orange cat" | DT 0 "gray dog" | WM 1 | DT 2 "pink dog" | DT 3 "black cat" | WM 3

- 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

  $$\textbf{codatatype}\ \mathit{llist} = (\mathrm{lnull}\colon \mathrm{LNil}) \mid \mathrm{LCons}\ (\mathrm{lhd}\colon {}'a)\ (\mathrm{ltl}\colon {}'a\ \mathit{llist})$$

- Selectors: *lhd*, *ltl*, Discriminator: *lnull*
- Examples:
  - LNil
  - $\mathrm{LCons}\ \mathit{1}\ (\mathrm{LCons}\ \mathit{2}\ (\mathrm{LCons}\ \mathit{3}\ \mathrm{LNil}))$
  - $\mathrm{LCons}\ \mathit{0}\ (\mathrm{LCons}\ \mathit{0}\ (\mathrm{LCons}\ \mathit{0}\ (\ldots)))$
- Coinduction principle for lazy list equality

  $$\begin{array}{l} R\ \mathit{lxs}\ \mathit{lys} \implies \\ (\bigwedge \mathit{lxs}\ \mathit{lys}.\ R\ \mathit{lxs}\ \mathit{lys} \implies \\ \quad \mathrm{lnull}\ \mathit{lxs} = \mathrm{lnull}\ \mathit{lys} \wedge \\ \quad (\neg\ \mathrm{lnull}\ \mathit{lxs} \longrightarrow \neg\ \mathrm{lnull}\ \mathit{lys} \longrightarrow \mathrm{lhd}\ \mathit{lxs} = \mathrm{lhd}\ \mathit{lys} \wedge R\ (\mathit{ctl}\ \mathit{lxs})\ (\mathit{ctl}\ \mathit{lys}))) \implies \\ \mathit{lxs} = \mathit{lys} \end{array}$$

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

Isabelle/HOL: Recursion and While Combinator

- Recursion

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

- While Combinator

```
definition while_option :: ('a ⇒ bool) ⇒ ('a ⇒ 'a) ⇒ 'a ⇒ 'a option where
while_option b c s = ...
```

  - 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

Isabelle/HOL: Corecursion

- Recursion: always eventually reduces an argument
- Corecursion: 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)

¬ lfinite lxs ⟹ lapp lxs lys = lxs
```

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 ⇒ 'a llist ⇒ bool where
  In_llist: in_llist x (LCons x lxs)
| Next_llist: in_llist x lxs ⇒ in_llist x (LCons y lxs)

in_llist 2 (LCons 1 (LCons (2 (. . . ))))
```

- Coinductive predicate
    - Infinite number of introduction rule applications

```
coinductive lprefix :: 'a llist ⇒ 'a llist ⇒ bool where
  LNil_lprefix: lprefix LNil lys
| LCons_lprefix: lprefix lxs lys ⇒ lprefix (LCons x lxs) (LCons x lys)

lprefix (LCons 1 (LCons (2 (. . . )))) (LCons 1 (LCons (2 (. . . ))))
```

Isabelle/HOL: (Co)inductive Predicates

- Inductive predicate
  - Finite number of introduction rule applications

```
inductive in_llist :: 'a ⇒ 'a llist ⇒ bool where
  In_llist: in_llist x (LCons x lxs)
| Next_llist: in_llist x lxs ⇒ in_llist x (LCons y lxs)

in_llist 2 (LCons 1 (LCons (2 (. . . ))))
```

- Coinductive predicate
  - Infinite number of introduction rule applications

```
coinductive lprefix :: 'a llist ⇒ 'a llist ⇒ bool where
  LNil_lprefix: lprefix LNil lys
| LCons_lprefix: lprefix lxs lys ⇒ lprefix (LCons x lxs) (LCons x lys)

lprefix (LCons 1 (LCons (2 (. . . )))) (LCons 1 (LCons (2 (. . . ))))
```

- Coinduction principle

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

[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 ⇒ ('o, 'i) op × 'o list))
```

  - Infinite trees: applying the selector `apply` "walks" a branch of the tree

```
corec count_op where count_op P n =
  Logic (λe. if P e then (count_op P (n + 1), [n+1]) else (count_op P n, []))
```

```
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

  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]

[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 $\mathsf{produce}_1$ *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′*))

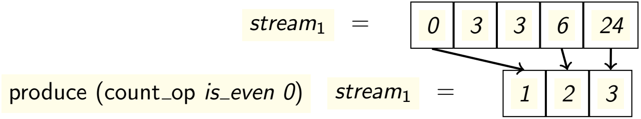
- lshift (@@) is a friend of corec function![2]
- $\mathsf{produce}_1$ has an induction principle based on the while invariant rule

[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 (λe. if P e then (count_op P (n + 1), [n+1]) else (count_op P n, []))
```



stream$_1$ = 0 | 3 | 3 | 6 | 24

produce (count_op *is_even* *0*) *stream*$_1$ = 1 | 2 | 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

```
datatype ('t::order, 'd) event = DT (tmp: 't) (data: 'd) | WM (wmk: 't)
```

Time-Aware Streams

- Time-Aware lazy lists

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

- 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]:

```
fun batches where batches [] tds = ([], tds)
| batches (wm # wms) tds = let (bs, tds') = batches wms [(t, d) ← tds. ¬ t ≤ wm]
                        in (DT wm [(t, d) ← tds. t ≤ wm] # bs, tds')

corec batch_op where
  batch_op tds = Logic (λev. case ev of DT t d ⇒ (batch_op (tds @ [(t, d)]), [])
  | WM wm ⇒ let (out, tds') = batches [wm] tds
            in (batch_op tds', [x ← out. data x ≠ []] @ [WM wm]))
```

```
mono lxs W ⟶ DT wm b ∈ lset (produce (batch_op tds) lxs) ⟶
(∀t ∈ set_t b. coll b t = lcoll lxs t + coll tds t) ∧
set_t b = batch_ts ((map (λ (t,d). DT t d) tds) @@ lxs) wm
```

```
mono lxs W ⟶ (∀t ∈ set_t tds. ∀wm ∈ W. ¬ t ≤ wm) ⟶ mono (produce (batch_op tds) 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