

Efficient and Verified Non-Terminating Programs with Isabelle-LLVM

Rafael Castro G. Silva

`razi@di.ku.dk`

Department of Computer Science
University of Copenhagen

23/11/2023

Introduction

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

- 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

Stream Processing

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

Stream Processing

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



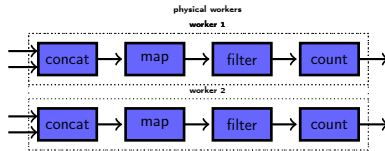
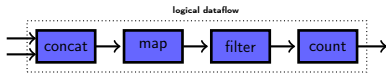
Cloud
DataFlow



Flink



- Highly Parallel



Time-Aware Stream Processing (part 1)

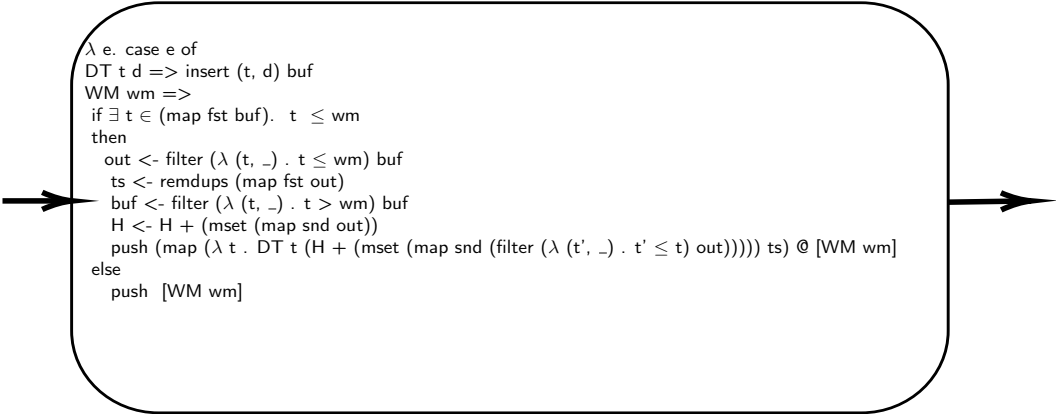
- Time-Aware Computations:
 - Timestamps: Metadata associating the data with some data collection
 - An unix timestamp
 - Version of the data
 - Logical grouping
 - Watermarks: Metadata indicating the completion of a data collection
 - e.g.: A watermark 5 says that there is no data associated with timestamp 5 or below arriving
 - Are increasingly monotonic (they don't go backwards in time)
 - e.g.:

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	...
--------------	--------------	--------------	----------	--------------	----------	--------------	--------------	--------------	----------	-----

Example of Time-Aware Stream Processing

DT 2 b	DT 1 a	WM 1	DT 2 c	DT 3 a	WM 4
--------	--------	------	--------	--------	------

buf = []
H = {}

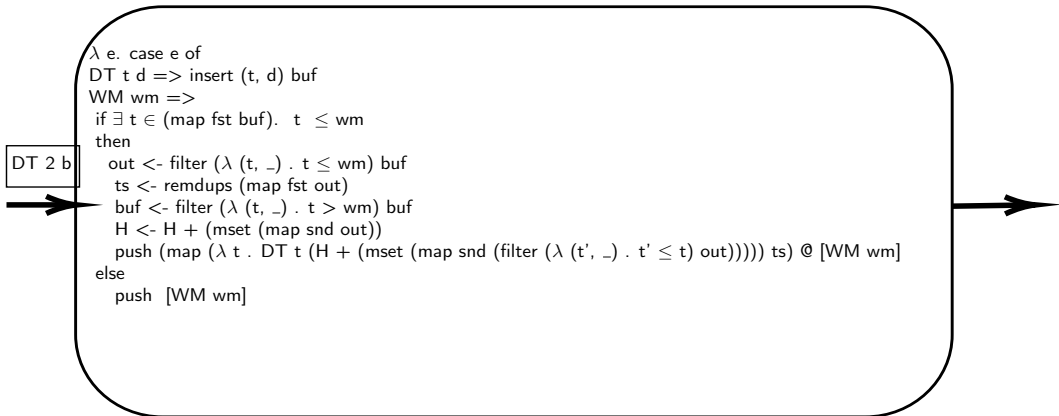


```
λ e. case e of
  DT t d => insert (t, d) buf
  WM wm =>
    if ∃ t ∈ (map fst buf). t ≤ wm
    then
      out <- filter (λ (t, _) . t ≤ wm) buf
      ts <- remdups (map fst out)
      buf <- filter (λ (t, _) . t > wm) buf
      H <- H + (mset (map snd out))
      push (map (λ t . DT t (H + (mset (map snd (filter (λ (t', _) . t' ≤ t) out))))) ts) @ [WM wm]
    else
      push [WM wm]
```

Example of Time-Aware Stream Processing

DT 1 a	WM 1	DT 2 c	DT 3 a	WM 4
--------	------	--------	--------	------

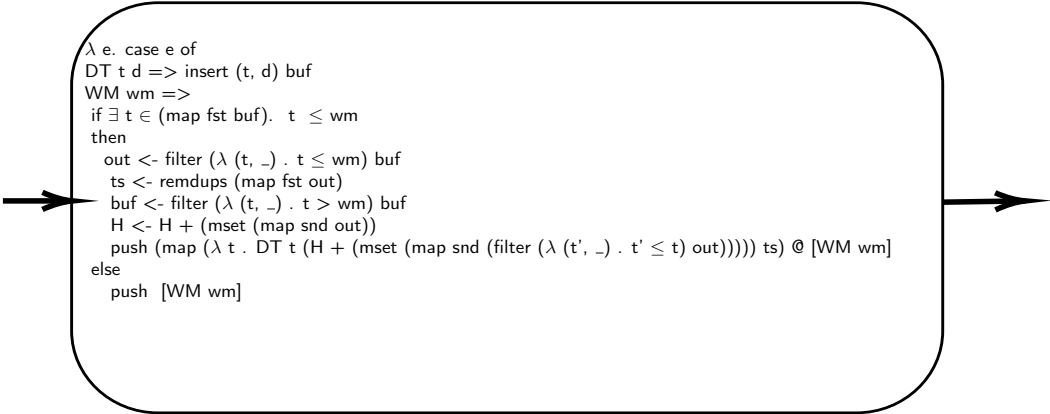
buf = []
H = {}



Example Time-Aware Stream Processing

DT 1 a	WM 1	DT 2 c	DT 3 a	WM 4
--------	------	--------	--------	------

buf = [(2,b)]
H = {}

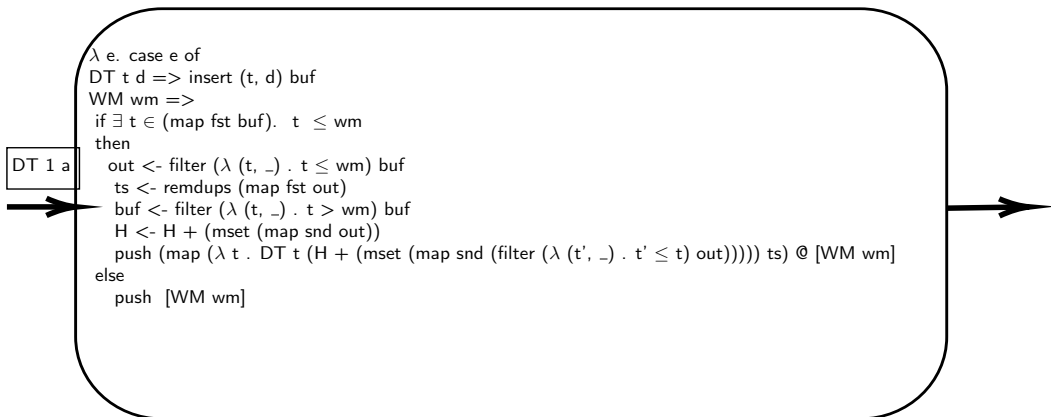


```
λ e. case e of
  DT t d => insert (t, d) buf
  WM wm =>
    if ∃ t ∈ (map fst buf). t ≤ wm
    then
      out <- filter (λ (t, _) . t ≤ wm) buf
      ts <- remdups (map fst out)
      buf <- filter (λ (t, _) . t > wm) buf
      H <- H + (mset (map snd out))
      push (map (λ t . DT t (H + (mset (map snd (filter (λ (t', _) . t' ≤ t) out))))) ts) @ [WM wm]
    else
      push [WM wm]
```

Example Time-Aware Stream Processing

WM 1	DT 2 c	DT 3 a	WM 4
------	--------	--------	------

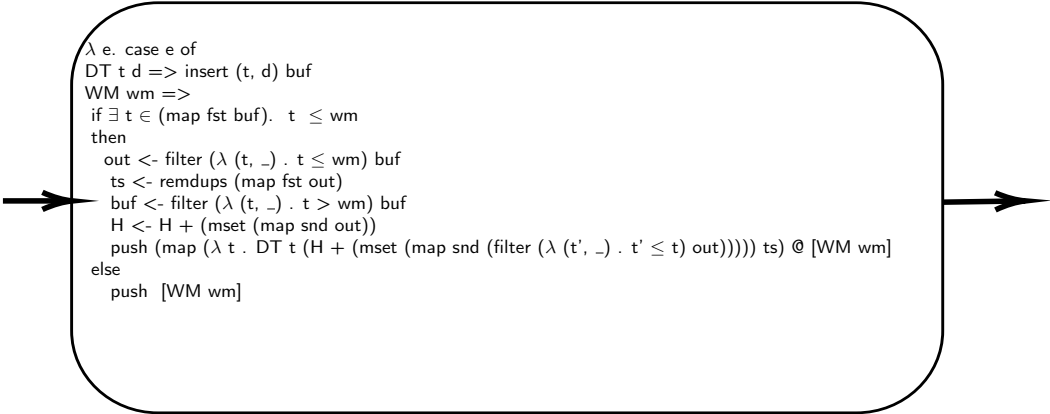
buf = [(2,b)]
H = {}



Example Time-Aware Stream Processing

WM 1	DT 2 c	DT 3 a	WM 4
------	--------	--------	------

buf = [(2,b), (1,a)]
H = {}

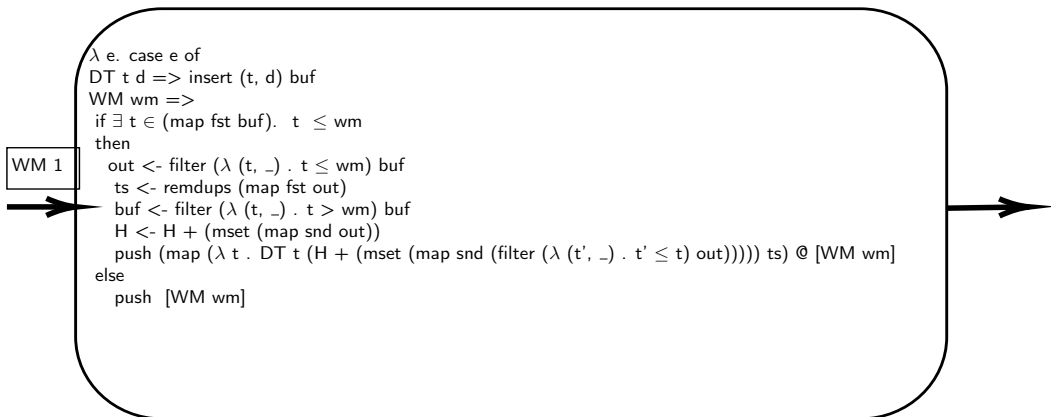


```
λ e. case e of
  DT t d => insert (t, d) buf
  WM wm =>
    if ∃ t ∈ (map fst buf). t ≤ wm
    then
      out <- filter (λ (t, _) . t ≤ wm) buf
      ts <- remdups (map fst out)
      buf <- filter (λ (t, _) . t > wm) buf
      H <- H + (mset (map snd out))
      push (map (λ t . DT t (H + (mset (map snd (filter (λ (t', _) . t' ≤ t) out))))) ts) @ [WM wm]
    else
      push [WM wm]
```

Example Time-Aware Stream Processing

DT 2 c	DT 3 a	WM 4
--------	--------	------

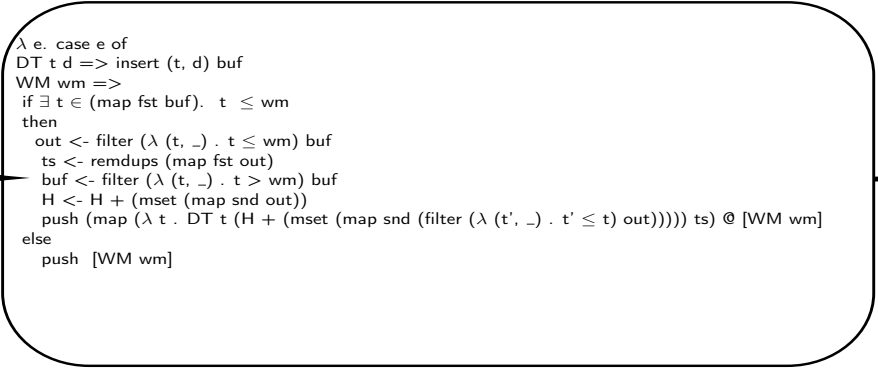
buf = [(2,b), (1,a)]
H = {}



Example Time-Aware Stream Processing

DT 2 c	DT 3 a	WM 4
--------	--------	------

buf = [(2,b)]
H = {a}



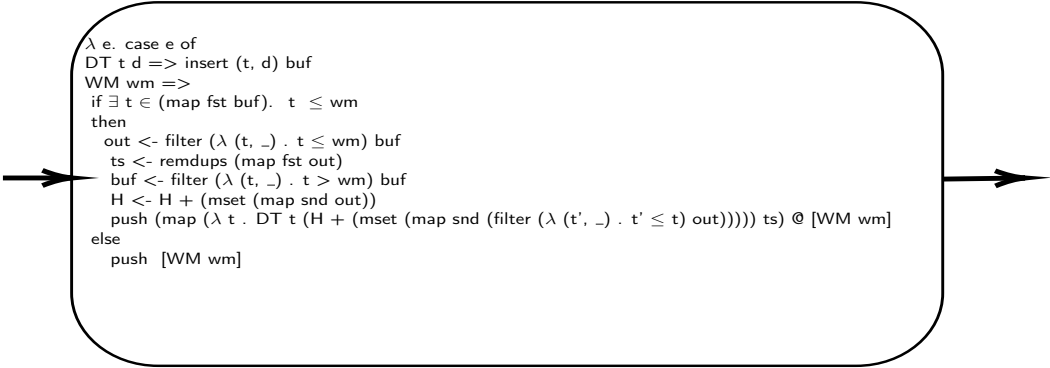
```
λ e. case e of
  DT t d => insert (t, d) buf
  WM wm =>
    if ∃ t ∈ (map fst buf). t ≤ wm
    then
      out <- filter (λ (t, _) . t ≤ wm) buf
      ts <- remdups (map fst out)
      buf <- filter (λ (t, _) . t > wm) buf
      H <- H + (mset (map snd out))
      push (map (λ t . DT t (H + (mset (map snd (filter (λ (t', _) . t' ≤ t) out))))) ts) @ [WM wm]
    else
      push [WM wm]
```

DT 1 {a}	WM 1
----------	------

Example Time-Aware Stream Processing

WM 4

buf = [(2,b), (2,c), (3,a)]
H = {a}

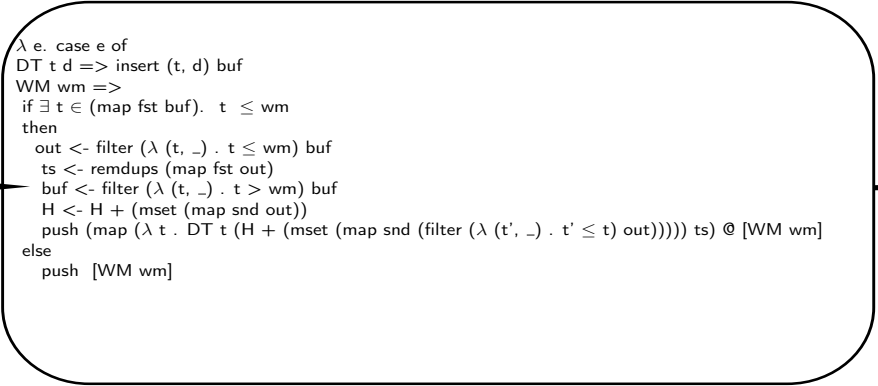


```
λ e. case e of
  DT t d => insert (t, d) buf
  WM wm =>
    if ∃ t ∈ (map fst buf). t ≤ wm
    then
      out <- filter (λ (t, _) . t ≤ wm) buf
      ts <- remdups (map fst out)
      buf <- filter (λ (t, _) . t > wm) buf
      H <- H + (mset (map snd out))
      push (map (λ t . DT t (H + (mset (map snd (filter (λ (t', _) . t' ≤ t) out))))) ts) @ [WM wm]
    else
      push [WM wm]
```

DT 1 {a} WM 1

Example Time-Aware Stream Processing

buf = []
H = {a,a,b,c}



```
λ e. case e of
  DT t d => insert (t, d) buf
  WM wm =>
    if  $\exists t \in (\text{map fst buf}). t \leq \text{wm}$ 
    then
      out <- filter ( $\lambda (t, \_) . t \leq \text{wm}$ ) buf
      ts <- remdups (map fst out)
      buf <- filter ( $\lambda (t, \_) . t > \text{wm}$ ) buf
      H <- H + (mset (map snd out))
      push (map ( $\lambda t . \text{DT } t (H + (\text{mset } (\text{map snd } (\text{filter } (\lambda (t', \_) . t' \leq t) \text{ out}))))$ ) ts) @ [WM wm]
    else
      push [WM wm]
```

DT 1 {a}	WM 1	DT 2 {a,b,c}	DT 3 {a,a,b,c}	WM 4
----------	------	--------------	----------------	------

Preliminaries

- Classical higher-order logic (HOL): Simple Typed Lambda Calculus + (Hilbert) axiom of choice + axiom of infinity + rank-1 polymorphism

- 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

- 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 (...)))

- Proofs by induction
- Proofs by coinduction

State of this work

What have I formalized so far? (part 1)

- Formalization stream processing (model)
 - Using Isabelle/HOL: (co)datatypes, (co)recursion, and (co)induction
 - Streams are lazy lists, and operators as a codatatype
 - Semantics: a `produce :: 'i llist \Rightarrow ('i, 'o) op \Rightarrow 'o llist` function that runs an operator throughout a lazy lists
 - Mix of recursion and corecursion: inductive and coinductive principles
 - Sequential composition
 - Correctness!

What have I formalized so far? (part 2)

- Time-Aware computations
 - Coinductive properties of streams: monotonicity and productivity
 - Building blocks operators:
 - Convenience operators: batching and incremental computations
 - Incremental computing: only update results that are affected by the new input
 - With verified properties: Soundness, Completeness, preservation of monotonicity, and preservation of productivity
 - Compositional reasoning
- Case studies with the building blocks:
 - Incremental histogram operator
 - Relational join

Next Steps

- It is executable! But slow!
 - Code generator: functional languages (OCaml, Haskell, SML...)
 - Functional data-structures (often not ideal)

- It is executable! But slow!
 - Code generator: functional languages (OCaml, Haskell, SML...)
 - Functional data-structures (often not ideal)
- How do we make efficient and verified programs in Isabelle/HOL?

- It is executable! But slow!
 - Code generator: functional languages (OCaml, Haskell, SML...)
 - Functional data-structures (often not ideal)
- How do we make efficient and verified programs in Isabelle/HOL?
- Isabelle-LLVM!
- Let's port this formalization to Isabelle-LLVM then!
- This is a non-terminating program

Isabelle-LLVM

- Isabelle Refinement Framework
 - Framework for step-wise refinement verification (refinement calculus): Specification \rightarrow Abstract Algorithm \rightarrow Less Abstract Algorithm \rightarrow Executable Code
 - Imperative HOL as backend (lowest layer in the refinement)
 - Shallow Embedding of Monadic programs in HOL
 - Separation Logic (heap memory reasoning)
- Isabelle-LLVM is a new backend for the Isabelle Refinement Framework
 - Generates LLVM code (efficient imperative code)

Isabelle Refinement Framework and Isabelle-LLVM

- Isabelle Refinement Framework
 - Framework for step-wise refinement verification (refinement calculus): Specification \rightarrow Abstract Algorithm \rightarrow Less Abstract Algorithm \rightarrow Executable Code
 - Imperative HOL as backend (lowest layer in the refinement)
 - Shallow Embedding of Monadic programs in HOL
 - Separation Logic (heap memory reasoning)
- Isabelle-LLVM is a new backend for the Isabelle Refinement Framework
 - Generates LLVM code (efficient imperative code)
- Can we write and verify non-terminating programs in this framework?

Isabelle Refinement Framework and Isabelle-LLVM

- Isabelle Refinement Framework
 - Framework for step-wise refinement verification (refinement calculus): Specification \rightarrow Abstract Algorithm \rightarrow Less Abstract Algorithm \rightarrow Executable Code
 - Imperative HOL as backend (lowest layer in the refinement)
 - Shallow Embedding of Monadic programs in HOL
 - Separation Logic (heap memory reasoning)
- Isabelle-LLVM is a new backend for the Isabelle Refinement Framework
 - Generates LLVM code (efficient imperative code)
- Can we write and verify non-terminating programs in this framework?
 - Yes and No!

- Knaster–Tarski theorem
 - Standard way to define the semantics of recursive definitions
 - Isabelle/HOL: Partial Function Package
 - Every monotonic function on Complete Chain Partial Order (CCPO) has a fixed point
 - Induction principle
- No need for well-foundness

What the Heck is a CCPO?

- Chain: A set in which all elements are comparable
- Complete Chain Partial Order:
 1. A partial order: `'a::order`
 2. A function that returns the supremum (least upper bound) from a chain `'a::order set \Rightarrow 'a`

What the Heck is a CCPO?

- Chain: A set in which all elements are comparable
- Complete Chain Partial Order:
 1. A partial order: `'a::order`
 2. A function that returns the supremum (least upper bound) from a chain `'a::order` set \Rightarrow `'a`
- Isabelle-LLVM's monad:

```
datatype 'a neM = SPEC (the_spec: 'a  $\Rightarrow$  bool) | FAIL
```

- Order: flat (every `SPEC` is greater than `FAIL`, `SPEC`s are only comparable when they are equal)
- Supremum from a chain: The `SPEC`, or the only `FAIL`
- Bottom: `FAIL`
 - Non-termination

The First Steps

Our CCPO Attempt

- Let's look in Isabelle!

The Other Steps

More Changes to Isabelle-LLVM?

- Separation Logic?
 - Express properties about the trace of the program
- Refinement Calculus?
- LLVM code generator?

Questions, comments and suggestions