Verified Time-Aware Stream Processing

Rafael Castro G. Silva

rasi@di.ku.dk

Department of Computer Science
University of Copenhagen

02/11/2023

What is this PhD/Status seminar about?

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

Contents

Introduction

Dataflow Models

- Stream Processing
- Dataflow Model
- Time-Aware Computations
- Bugs in Stream Processing

Preliminaries

Isabelle/HOL

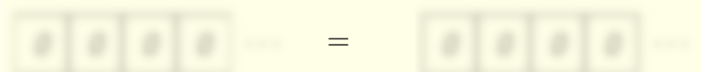
- HOL
- Isabelle/HOL

Isabelle/HOL: (Co)datatypes

- Datatypes and Codatatypes

$$\textbf{codatatype}\ (\mathsf{lset}\colon {'a})\ \mathit{llist} = \mathsf{lnull}\colon \mathsf{LNil} \mid \mathsf{LCons}\ (\mathsf{lhd}\colon {'a})\ (\mathsf{ltl}\colon {'a}\ \mathit{llist})$$
$$\textbf{for}\ \mathsf{map}\colon \mathsf{lmap}\ \textbf{where}\ \mathsf{ltl}\ \mathsf{LNil} = \mathsf{LNil}$$

- Examples:
  - LNil
  - LCons $\mathit{1}$ (LCons $\mathit{2}$ (LCons $\mathit{3}$ LNil))
  - LCons $\mathit{0}$ (LCons $\mathit{0}$ (LCons $\mathit{0}$ ($\ldots$)))
- Induction principle assuming membership in the lazy list
- Coinductive principle for lazy list equality:
  - Show that there is a pair of goggles that makes them to look the same, which implies that:
    - The first lazy list if empty iff second is
    - They have the same head
    - Their tail looks the same

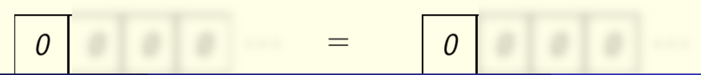


Isabelle/HOL: (Co)datatypes

- Datatypes and Codatatypes

$$\textbf{codatatype}\ (\mathsf{lset}\colon {'a})\ \mathit{llist} = \mathsf{lnull}\colon \mathsf{LNil} \mid \mathsf{LCons}\ (\mathsf{lhd}\colon {'a})\ (\mathsf{ltl}\colon {'a}\ \mathit{llist})$$
$$\textbf{for}\ \mathsf{map}\colon \mathsf{lmap}\ \textbf{where}\ \mathsf{ltl}\ \mathsf{LNil} = \mathsf{LNil}$$

- Examples:
  - LNil
  - LCons *1* (LCons *2* (LCons *3* LNil))
  - LCons *0* (LCons *0* (LCons *0* (. . .)))
- Induction principle assuming membership in the lazy list
- Coinductive principle for lazy list equality:
  - Show that there is a pair of goggles that makes them to look the same, which must imply that:
    - The first lazy list if empty iff second is
    - They have the same head
    - Their tail looks the same



Isabelle/HOL: (Co)datatypes

- Datatypes and Codatatypes

> **codatatype** (lset: *'a*) *llist* = lnull: LNil | LCons (lhd: *'a*) (ltl: *'a llist*)
> **for** map: lmap **where** ltl LNil = LNil

- Examples:
  - LNil
  - LCons *1* (LCons *2* (LCons *3* LNil))
  - LCons *0* (LCons *0* (LCons *0* (...)))
- Induction principle assuming membership in the lazy list
- Coinductive principle for lazy list equality:
  - Show that there is a pair of goggles that makes them to look the same, which must imply that:
    - The first lazy list if empty iff second is
    - They have the same head
    - Their tail looks the same

$$\begin{array}{|c|c|c|c|}\hline 0 & 0 & 0 & 0 \\ \hline\end{array}\cdots \quad = \quad \begin{array}{|c|c|c|c|}\hline 0 & 0 & 0 & 0 \\ \hline\end{array}\cdots$$

Isabelle/HOL: Recursion and While Combinator

- Recursion

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

- While Combinator

$$\textbf{definition}\ \mathsf{while\_option} :: ({'a} \Rightarrow \mathit{bool}) \Rightarrow ({'a} \Rightarrow {'a}) \Rightarrow {'a} \Rightarrow {'a}\ \mathit{option}\ \textbf{where}$$
$$\mathsf{while\_option}\ b\ c\ s = \ldots$$

- While rule for invariant reasoning:
  - There is something that holds before a step; that thing still holds after the step

Isabelle/HOL: Corecursion and Friends

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

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

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

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

lconcat *lxs* = case *lxs* of LNil ⇒ LNil | LCons *xs lxs'* ⇒ lshift *xs* (lconcat *lxs'*)

Isabelle/HOL: (Co)inductive Predicates

- Inductive predicate
  - Finite number of introduction rule applications

inductive in_llist $:: {'a} \Rightarrow {'a}\ \mathit{llist} \Rightarrow \mathit{bool}$ **where**
  In_llist: in_llist x (LCons x lxs)
| Next_llist: in_llist x lxs $\Rightarrow$ in_llist x (LCons y lxs)

in_llist $(2{::}nat)$ (LCons 1 (LCons $(2\ (\ldots))))$

- Coinductive predicate
  - Infinite number of introduction rule applications

coinductive lprefix $:: {'a}\ \mathit{llist} \Rightarrow {'a}\ \mathit{llist} \Rightarrow \mathit{bool}$ **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 $(2\ (\ldots))))$

- Coinduction principle

Lazy Lists Processors

Operators

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

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

  - Infinite trees: applying the selector apply "walks" a branch of the tree
- Produce function: applies the logic (co)recursively throughout a lazy list

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

Operators: Example

- 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, []))

$stream_2$ = $\boxed{0}\,\boxed{3}\,\boxed{3}\,\boxed{6}\,\boxed{24}$

produce (count_op is_even 0) $stream_2$ = $\boxed{1}\,\boxed{2}\,\boxed{3}$

Sequential Composition

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

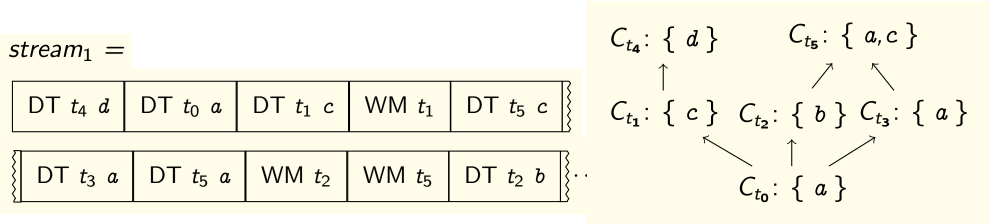
$$
\begin{array}{l}
\textbf{definition}\ \text{fproduce}\ op\ xs = \text{fold}\ (\lambda e\ (op, out).\\
\quad \texttt{let}\ (op', out') = \text{apply}\ op\ e\ \texttt{in}\ (op', out\ @\ out'))\ xs\ (op, [])\\
\\
\textbf{corec}\ \text{comp\_op}\ \textbf{where}\\
\quad \text{comp\_op}\ op_1\ op_2 = \text{Logic}\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

- Skip n

Time-Aware Operators

Monotone and Productive Time-Aware Streams

- Monotone
- Productive



(a) Prefix of $stream_1$ (b) Corresponding set of collections

Figure: An example stream and its collections (ordered by their time-stamps)

Building Blocks: Batch Operator

Batch Operator: Soundness

Batch Operator: Completeness

- Uses soundness of `batch_op`
- Proof by induction over n

$$\begin{aligned}
&\text{mono_prod } \mathit{lxs}\ W \longrightarrow (\exists i\ d.\ \text{enat } i < \text{llength } \mathit{lxs} \wedge \text{lnth } \mathit{lxs}\ i = \text{DT } t\ d \wedge n = \text{Suc } i) \vee \\
&n = 0 \wedge t \in \text{set_t } \mathit{buf} \longrightarrow (\forall t' \in \text{set_t } \mathit{buf}.\ \text{lfinite } \mathit{lxs} \vee \exists wm \geq t'\ .\ \text{WM } wm \in \text{lset } \mathit{lxs}) \longrightarrow \\
&\exists wm\ \mathit{batch}.\ \text{DT } wm\ \mathit{batch} \in \text{lset } (\text{produce } (\text{batch_op } \mathit{buf})\ \mathit{lxs}) \wedge t \in \text{set_t } \mathit{batch} \vee \\
&(\forall k \in \{n\ ..{<}\ \text{the_enat } (\text{llength } \mathit{lxs})\}\ .\ \neg\ (\exists t' \geq t.\ \text{lnth } \mathit{lxs}\ k = \text{WM } t')) \wedge \text{lfinite } \mathit{lxs}
\end{aligned} \tag{1}$$

Batch Operator: Monotone

Batch Operator: Productive

Building Blocks: Incremental Operator

Batch Operator: Soundness

Batch Operator: Completeness

Batch Operator: Monotone

Batch Operator: Productive

Compositional Reasoning

Case Study

Histogram

Histogram: Soundness

Histogram: Completeness

Histogram: Monotone

Histogram: Productive

Efficient Histogram

- Foo

Join

Join: Soundness

Join: Completeness

Join: Monotone

Next Steps

Next Steps

Questions, comments and suggestions