delta
Ordered Types for Stream Processing

Joseph W. Cutler
Christopher Watson
Philip Hilliard
Harrison Goldstein
Caleb Stanford (UC Davis)
Benjamin C. Pierce (University of Pennsylvania)
delta
Ordered Types for Stream Processing

Joseph W. Cutler
Christopher Watson
Philip Hilliard
Harrison Goldstein
Caleb Stanford (UC Davis)
Benjamin C. Pierce (University of Pennsylvania)

January 10, 2024
Stream Processing
What is a Stream?
What is a Stream?

An *ordered* sequence

![Diagram of ordered sequence](image-url)
What is a Stream?

An ordered sequence arriving incrementally
What is a Stream?

An ordered sequence
arriving incrementally
and possibly unbounded in size.
What is a Stream?

An ordered sequence
arriving incrementally
and possibly unbounded in size.
What is Stream Processing?
What is Stream Processing?

2 → sum → 2
What is Stream Processing?
What is Stream Processing?
What is Stream Processing?

2 → 1 → sum → 2 → 3
What is Stream Processing?
What is Stream Processing?
What is Stream Processing?
What is Stream Processing?
What is Stream Processing?
What is Stream Processing?

[2, 1, 4] → sum → [2, 3, 7]

Incremental List Processing
What is Stream Processing?

Incremental List Processing
(with bounded memory)
Programming Models for Streams
Programming Models for Streams

Manual State Machine

's -> 'a -> 's * ('b list)

❌ Low-Level
Programming Models for Streams

Manual State Machine

's -> 'a -> 's * ('b list)

❌ Low-Level

Functional Reactive Programming

time -> 'a

✅ High-Level

❌ Not Resource-Aware
Programming Models for Streams

Manual State Machine

\[ 's \rightarrow 'a \rightarrow 's \times ('b \text{ list}) \]

\[ \text{x} \text{ Low-Level} \]

Functional Reactive Programming

\[ \text{time} \rightarrow 'a \]

\[ \checkmark \text{ High-Level} \quad \text{x} \text{ Not Resource-Aware} \]

Streaming eDSL

\[ \text{map : ('a -> 'b) -> 'a stream -> 'b stream} \]

\[ \text{filter : ('a -> 'b) -> 'a stream -> 'b stream} \]

\[ \text{window : ('a list -> 'b) -> int -> 'a stream -> 'b stream} \]

\[ \text{fold : ('a -> 'b -> 'b) -> 'b -> 'a stream -> 'b stream} \]

\[ \checkmark \text{ Resource Aware} \]

\[ \checkmark \text{ High-Level} \]

\[ \text{x} \text{ Not Expressive} \]
What if I told you...

You can **write** a natural function on lists...

```
fun partialSums(acc : int, xs : int list) : int list =
    case xs of
    | nil => acc
    | y::ys => let acc′ = y + acc in
              acc′::partialSums(acc′,ys)
```

... and **run** it as a streaming program?
Familiar functional syntax…

fun foo(xs : int list) =
    case xs of
        nil => …
        y :: ys => … foo(ys) …

… with streaming semantics!

✅ Expressive    ✅ High-Level
Familiar functional syntax...

```
fun foo(xs : int list) =
  case xs of
    nil => ...
    y :: ys => ... foo(ys) ...
```

... with streaming semantics!

✅ Expressive  ✅ High-Level

And static prevention of space leaks

```
fun foo(xs : int list) = ...
```

?  Stateless  Bounded State  Unbounded State

✅ Resource-Aware
How?
How?

With an *Ordered* Substructural Type System!
A Non-Example, for Inspiration

fun reverse(xs : s*) : s* =
  case xs of
    nil => nil
  | y :: ys => snoc(reverse(ys);y)

“Stream” is written with a star in delta, like regular expressions
A *Non*-Example, for Inspiration

fun reverse(xs : s*) : s* =
case xs of
  nil => nil
| y :: ys => snoc(reverse(ys);y)

| | | | | |
y arrives before ys...
A Non-Example, for Inspiration

fun reverse(xs : s*) : s* =
  case xs of
   nil => nil
 | y :: ys => snoc(reverse(ys);y)

y arrives before ys...

... but is sent after

“Stream” is written with a star in delta, like regular expressions
A *Non*-Example, for Inspiration

fun reverse(xs : s*) : s* =
  case xs of
    nil => nil
  | y :: ys => snoc(reverse(ys);y)

We have to save the *entire stream* in memory!

“Stream” is written with a star in delta, like regular expressions

<table>
<thead>
<tr>
<th>y arrives before ys...</th>
</tr>
</thead>
</table>

... but is sent after

|-------------------------|
Core Idea:

If all variables are used *in order of arrival of the corresponding data*, then the program is streamable with no auxiliary memory.
Core Idea:

If all variables are used in order of arrival of the corresponding data, then the program is streamable with no auxiliary memory.

delta is stateless by default

(programmers can selectively introduce state when needed)
Typing Judgment

\[ \Gamma \vdash e : S \]

- **Input Types**
- **Stream Program**
- **Output Type**
Typing Judgment

\[ \Gamma \vdash e : S \]

\[ \Gamma = (x_0 : S_0); (x_1 : S_1); \ldots; (x_n : S_n) \]

Input Types | Stream Program | Output Type

First variable to arrive | Last variable to arrive
(Questions?)
Variables

\[ \Gamma; (x : S); \Gamma' \vdash x : S \]
Variables

\[ \Gamma; (x : S) ; \Gamma' \vdash x : S \]
Variables

\[ \Gamma; (x : S); \Gamma' \vdash x : S \]
Variables

\[
\Gamma; (x : S); \Gamma' \vdash x : S
\]
Variables

\[ \Gamma; (x : S) ; \Gamma' \vdash x : S \]

\[ \Gamma \quad S \quad \Gamma' \]

\[ S \quad x \]
Nil and Cons

\[ \Gamma \vdash \text{nil} : S^* \]
Nil and Cons

\[ \Gamma \vdash \text{nil} : S^* \]
Nil and Cons

\[ \Gamma \vdash \text{nil} : S^* \]
Nil and Cons

\[
\frac{\Gamma \vdash \text{nil} : S^*}{\Gamma; \Delta \vdash e::e' : S^*}
\]

\[
\frac{\Gamma \vdash e : S \quad \Delta \vdash e' : S^*}{\Gamma; \Delta \vdash e::e' : S^*}
\]
Nil and Cons

\[ \Gamma \vdash \text{nil} : S^* \]

**Early inputs used for first output**

\[ \Gamma \vdash e : S \]

\[ \Delta \vdash e' : S^* \]

\[ \Gamma; \Delta \vdash e;::e' : S^* \]

**Later inputs used for rest of outputs**

\[ \Gamma \]

\[ \text{nil} \]

\[ \text{EOS} \]
Nil and Cons

\[
\Gamma \vdash \text{nil} : S^* \\
\Gamma \vdash e : S \\
\Gamma ; \Delta \vdash e :: e' : S^*
\]

Early inputs used for first output

Later inputs used for rest of outputs

\[
\Gamma \vdash e : S \\
\Delta \vdash e' : S^*
\]

\[
\Gamma ; \Delta \vdash e :: e' : S^*
\]

\[
e :: e'
\]
Nil and Cons

\[ \Gamma \vdash \text{nil} : S^* \]

**Early inputs used for first output**

\[ \Gamma \vdash e : S \]
\[ \Delta \vdash e' : S^* \]
\[ \Gamma ; \Delta \vdash e :: e' : S^* \]

**Later inputs used for rest of outputs**

\[ \Gamma \vdash \text{nil} : S^* \]

\[ \Gamma \vdash e : S \]
\[ \Delta \vdash e' : S^* \]
\[ \Gamma ; \Delta \vdash e :: e' : S^* \]

\[ \Gamma \xrightarrow{\text{nil}} \quad \text{EOS} \]

\[ \Gamma \xrightarrow{e :: e'} \quad S \]
Nil and Cons

\[ \Gamma \vdash \text{nil} : S^* \]

Early inputs used for first output

Later inputs used for rest of outputs

\[ \Gamma \vdash e : S \quad \Delta \vdash e' : S^* \]

\[ \Gamma ; \Delta \vdash e :: e' : S^* \]
**Star-Case**

\[
\Gamma; \Gamma' \vdash e : T \quad \Gamma; (y : S); (ys : S^*); \Gamma' \vdash e' : T
\]

\[
\Gamma; (xs : S^*); \Gamma' \vdash \text{case}(xs, e, y.ys.e') : T
\]
Filter

Looks like **list**
filter, runs like **stream filter**!

fun filter[s](p : s -> Bool)(xs : s*) : s* =
case xs of
  nil => nil
  | y::ys => let zs = filter(p)(ys) in
    if p(y) then y::zs else zs
fun filter[s](p : s -> Bool)(xs : s*) : s* =
  case xs of
    nil => nil
  | y::ys => let zs = filter(p)(ys) in
    if p(y) then y::zs else zs

We can recover all the standard list combinators as
stream combinators in this style!
But Wait, There’s More!
Ordered Structure Yields Rich Stream Types

$$S \cdot T$$

Concatenation Type
Ordered Structure Yields Rich Stream Types

\[ S + T \]
Sum Type

\[ S \cdot T \]
Concatenation Type
Ordered Structure Yields Rich Stream Types

\[ S + T \]  
Sum Type

\[ S \cdot T \]  
Concatenation Type

Int  
Singleton Type
Ordered Structure Yields Rich Stream Types

\[ S + T \]
Sum Type

\[ S \cdot T \]
Concatenation Type

\[ \text{Int} \]
Singleton Type

\[ \varepsilon \]
Empty Stream Type
What streams do these types describe?

\[ \text{Int} + \text{Int} \cdot \text{Int} \quad \text{Int} \cdot (\text{Int} + \varepsilon) \]
Stream Types vs. Separation Logic

$S \cdot T$
Separation in time

$S \ast T$
Separation in (heap) space

Technical footnote:
Rich types let us build stream programs compositionally
Example

Compute the average of each run above 3
Example

Compute the average of each run above 3
Example

Compute the average of each run above 3
Compute the average of each run above 3
Example

Compute the average of each run above 3
Example

Compute the average of each run above 3
Example

Compute the average of each run above 3
Example

Compute the average of each run above 3
Example

Compute the average of each run above 3
In Flink:

“Compute averages of runs above 3”
In Flink:

“Compute averages of runs above 3”

```scala
xs.flatMapWithState((x : Int, st : Option[Int,Int]) =>
  st match {
    case None => if x > 3 then ([],Some(1,x)) else ([],None)
    case Some(num, tot) =>
      if x > 3 then
        ([],Some(num + 1, tot + x))
      else
        ([(x + tot) / (num + 1)], None)
  })
```

In Flink:

"Compute averages of runs above 3"

```scala
xs.flatMapWithState((x, st) =>
  st match {
    case None => if x > 3 then ((), Some(1, x)) else ((), None)
    case Some(num, tot) =>
      if x > 3 then
        ((), Some(num + 1, tot + x))
      else
        ([(x + tot) / (num + 1)], None)
  }
)
```
In delta:

delta gives you **types** to help think about the problem!

\[
\text{Int} \cdot \text{Int}^* \\
\text{“Nonempty run of Ints”}
\]

\[
(\text{Int} \cdot \text{Int}^*)^* \\
\text{“Stream of nonempty runs of Ints”}
\]
Typeful Programming in delta

\[
\text{Int} \cdot \text{Int}^* \xrightarrow{\text{avgRun}} \text{Int}
\]

\[
\text{Int}^* \xrightarrow{\text{parseRuns}} (\text{Int} \cdot \text{Int}^*)^* \xrightarrow{\text{map(avgRun)}} \text{Int}^*
\]

avgAbove3
fun average1Run(w : Int . Int*) : Int =
let (x;xs) = w in
let (k,n) = (sum(xs), length(xs)) in
(k + x) / (n + 1)

fun avgAbove3(XS : Int*) : Int* =
map(average1Run)(parseRuns(geq3)(xs))
fun average1Run(w : Int . Int*) : Int =
let (x;xs) = w in
let (k,n) = (sum(xs), length(xs)) in
(k + x) / (n + 1)

fun avgAbove3(XS : Int*) : Int* =
map(average1Run)(parseRuns(geq3)(xs))

*Exact same semantics as the fold in Flink!*
But Wait, There’s *Even* More!
What about multiple parallel inputs?
What about multiple parallel inputs?
What about multiple parallel inputs?

\[ S_1 \parallel S_2 \parallel S_3 \rightarrow \quad \text{Block} \quad \rightarrow \quad T_1 \parallel T_2 \]

Multiple inputs are products!
“Parallel Substreams” Type

\[ S \parallel T \]

"A stream of type S interleaved with a stream of type T”
(with elements tagged to indicate their source)
Parallel inputs risk nondeterminism!
Problem: Multiple Equivalent Interleavings

\[ \text{Int}^* \parallel \text{Char}^* \]
Problem: Multiple Equivalent Interleavings

\[ \text{Int}^* \parallel \text{Char}^* \]

Streams with parallel are actually *partially ordered*
We don’t want to be able to write this:

```kotlin
fun imposeOrder(x : Int || Char) : Int =
<... if the Int arrives first,
   send it along and drop the Char;
if the Char arrives first,
   send 42 and drop the Int...>
```
Core Idea 2:

If variables corresponding to unordered parts of the data are never used in an ordered way, the program is deterministic.
Core Idea 2:

If variables corresponding to unordered parts of the data are never used in an ordered way, the program is deterministic.

delta is deterministic by default

(programmers can selectively introduce nondeterminism)
Parallel Right Rule

\[
\frac{\Gamma \vdash e : S \quad \Gamma \vdash e' : T}{\Gamma \vdash (e, e') : S \parallel T}
\]
Parallel Right Rule

\[ \Gamma \vdash e : S \quad \Gamma \vdash e' : T \]
\[ \Gamma \vdash (e, e') : S \parallel T \]
Parallel Right Rule

\[
\frac{
\Gamma \vdash e : S \quad \Gamma \vdash e' : T
}{
\Gamma \vdash (e, e') : S \parallel T
}\]
Parallel Left Rule, Take 1

\[
\frac{\Gamma; \lambda \lambda; \Gamma' \vdash e : R \quad \Gamma; z : S \parallel T; \Gamma' \vdash \text{let}(x,y) = z \text{ in } e : R}{\Gamma; \lambda \lambda; \Gamma' \vdash e : R}
\]
Parallel Left Rule, Take 1

\[
\frac{
\Gamma; z : S \| T; \Gamma' \vdash e : R
}{
\Gamma; ???; \Gamma' \vdash \text{let}(x, y) = z \text{ in } e : R
}
\]

\[(x : S); (y : T)\]
Parallel Left Rule, Take 1

\[
\Gamma; ???; \Gamma' \vdash e : R
\]

\[
\Gamma; z : S \| T; \Gamma' \vdash \text{let} (x, y) = z \text{ in } e : R
\]

\[
(x : S \times (y : T))
\]
Parallel Left Rule, Take 1

\[
\frac{\Gamma; ???; \Gamma' \vdash e : R}{\Gamma; z : S \parallel T; \Gamma' \vdash \text{let } (x, y) = z \text{ in } e : R}
\]

\( (x : S \times (y : T)) \quad \text{and} \quad (y : T); (x : S) \)
Parallel Left Rule, Take 1

\[
\frac{\Gamma; ???; \Gamma' \vdash e : R}{\Gamma; z : S \parallel T; \Gamma' \vdash \text{let} (x, y) = z \text{ in } e : R}
\]

\[(x : S \times (y : T)) \quad \text{and} \quad (y : T \times (x : S))\]
Parallel Left Rule, Take 1

\[
\Gamma; ???; \Gamma' \vdash e : R \\
\Gamma; z : S \parallel T; \Gamma' \vdash \text{let} (x, y) = z \text{ in } e : R
\]
“Bunched” Contexts

\[ \Gamma ::= \cdot \mid x : S \mid \Gamma, \Gamma \mid \Gamma; \Gamma \]
“Bunched” Contexts

\[ \Gamma ::= \cdot | x : S | \Gamma, \Gamma | \Gamma; \Gamma \]

\[ \Gamma, \Delta \]

Unordered

\[ \Gamma; \Delta \]

Ordered
"Bunched" Contexts

\[ \Gamma ::= \cdot \mid x : S \mid \Gamma, \Gamma \mid \Gamma; \Gamma \]

\[ \Gamma, \Delta \]

Unordered
Corresponds to parallel

\[ \Gamma; \Delta \]

Ordered
Corresponds to concat
Unordered data can’t be used in an ordered way

\[ (x : S), (y : T) \vdash (x; y) : S \cdot T \]

Formally: Cat-R rule requires semicolon context
Unordered data can’t be used in an ordered way

\[
\begin{align*}
(x : S), (y : T) &\quad \text{\textbf{x}} \quad (x ; y) : S \cdot T
\end{align*}
\]

Formally: Cat-R rule requires semicolon context
A Stream Partitioner

Looks like **list** partition, runs like **stream** partition!

```
fun partition[s](p : s->Bool)(xs : s*) : s*||s* =
    case xs of
    nil => nil
    | y::ys => let (zs,ws) = partition(p)(ys) in
        if p(y) then (y::zs,ws) else (zs,y::ws)
```
A Stream Partitioner

Looks like **list** partition, runs like **stream** partition!

```plaintext
fun partition[s](p : s->Bool)(xs : s*) : s* || s* =
    case xs of
        nil => nil
    | y::ys => let (zs,ws) = partition(p)(ys) in
                 if p(y) then (y::zs,ws) else (zs,y::ws)
```

Again, we can recover many standard parallelism combinators as stream programs in this style!
Deterministic Merging of Parallel Streams

fun merge[s](xs : s*, ys : s*) : (s || s)* =
case xs of
  nil => nil
| x'::xs' =>
  case ys of
    nil => nil
  | y'::ys' => (x',y') :: merge(xs',ys')
Execution model
Today

Single-node Haskell interpreter

Soon

Compiler to Rust-based runtime above Hydro dataflow engine
Single-Node Compiler

fun foo(x : s) : t = ...

Multi-Node Compiler

fun foo(x : s) : t = 
let (y,z) = (e1,e2)
in e3
Wrapping up...
Future Directions

Compile Targets

- Hydro
- Hardware
- Streaming DBs

Case Studies

- IoT and Edge
- Financial Data
- ML Streaming

Theory

- Denotational Semantics
- FRP Connections
- Rewriting & Optimization

<Your Application Here>
Thank you!!

jwc@seas.upenn.edu
bcpierce@seas.upenn.edu

https://www.seas.upenn.edu/~jwc/assets/stream-types.pdf

https://github.com/alpha-convert/delta

Questions?