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
TFP

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)

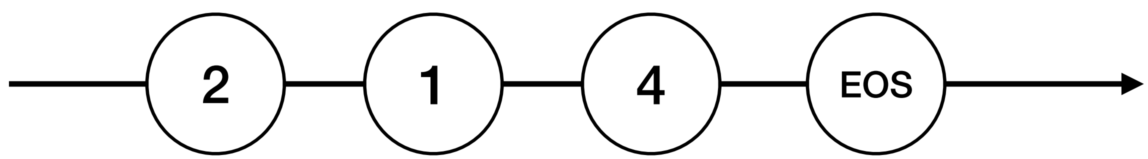
January 10, 2024
TFP

Stream Processing

What is a Stream?

What is a Stream?

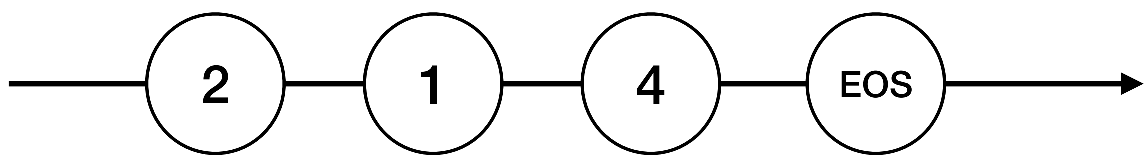
An ordered sequence



What is a Stream?

An ordered sequence

arriving incrementally

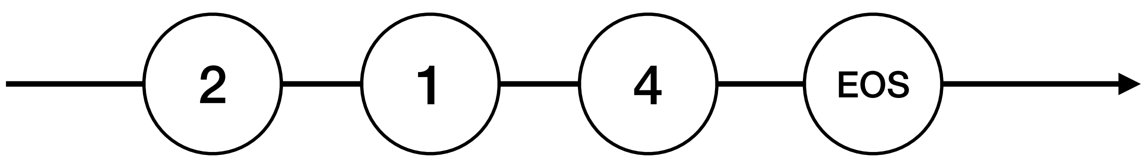


What is a Stream?

An ordered sequence

arriving incrementally

and possibly unbounded in size.

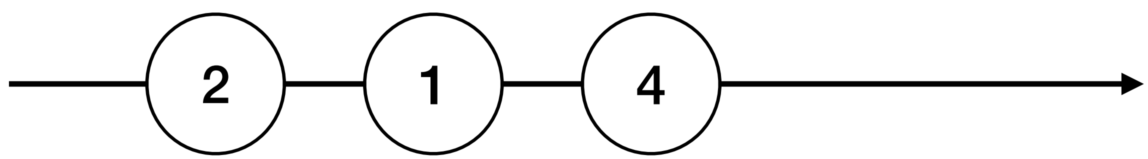


What is a Stream?

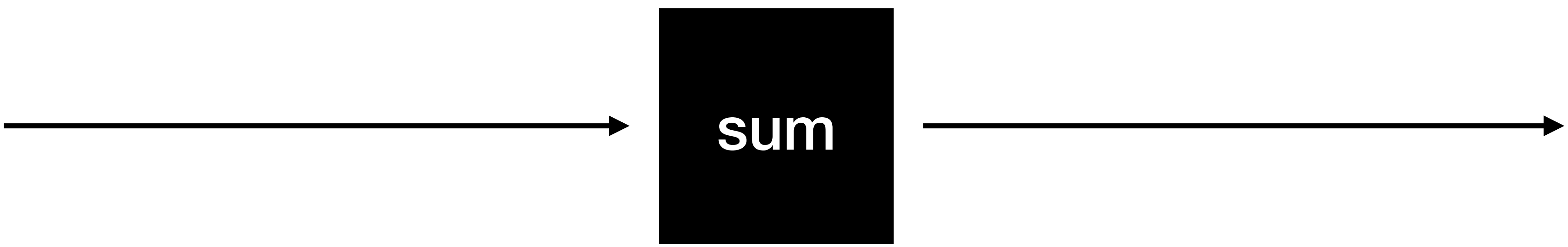
An ordered sequence

arriving incrementally

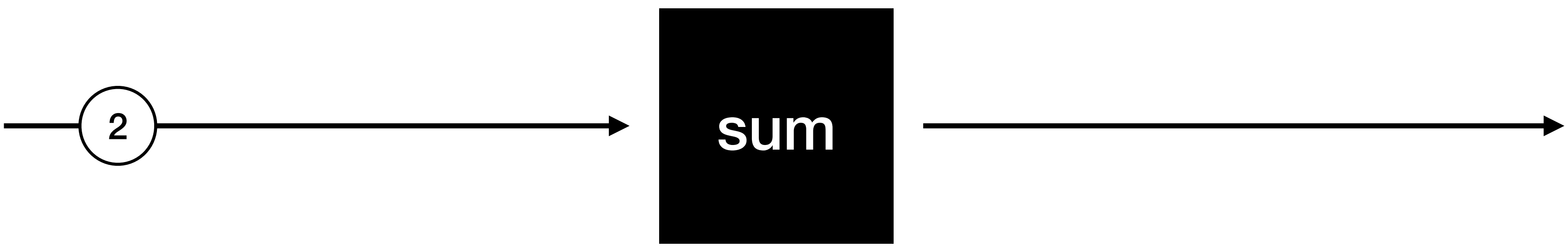
and possibly unbounded in size.



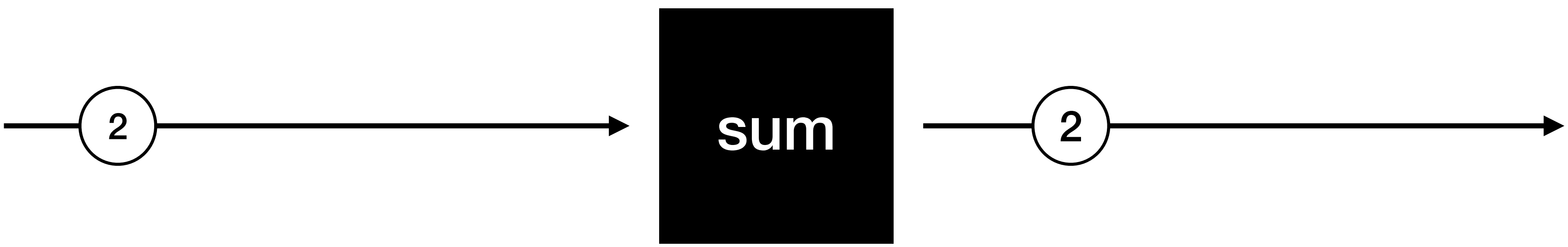
What is Stream Processing?



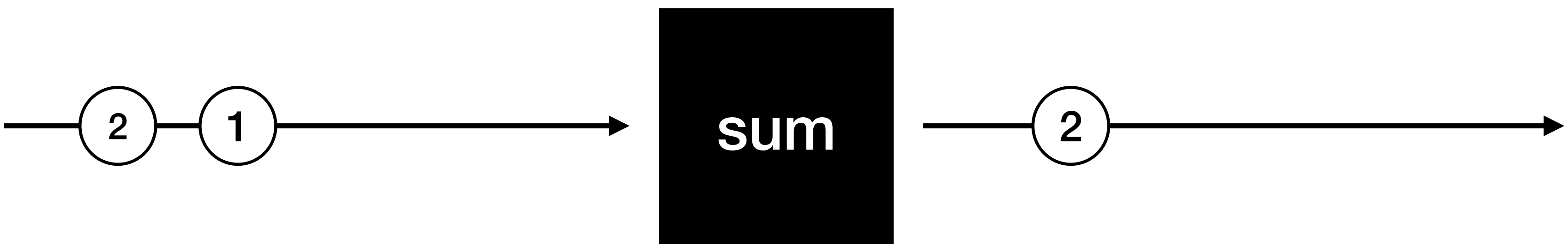
What is Stream Processing?



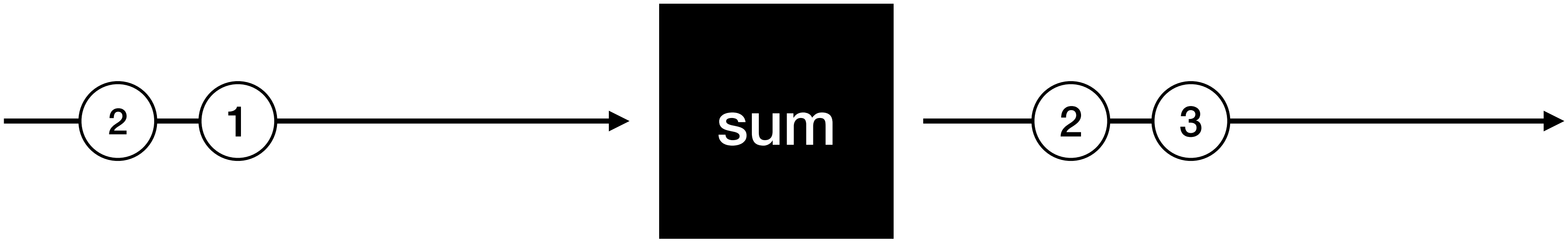
What is Stream Processing?



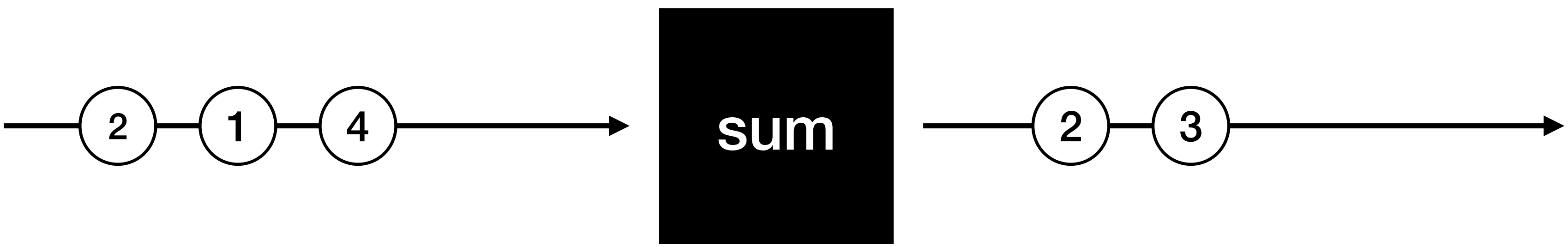
What is Stream Processing?



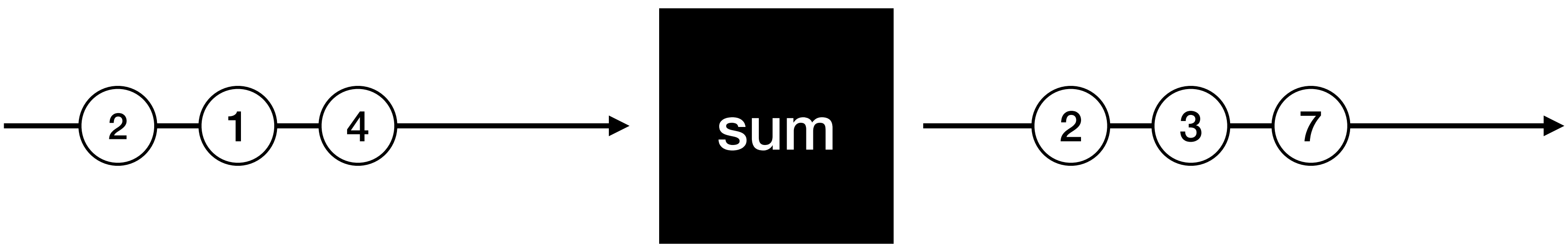
What is Stream Processing?



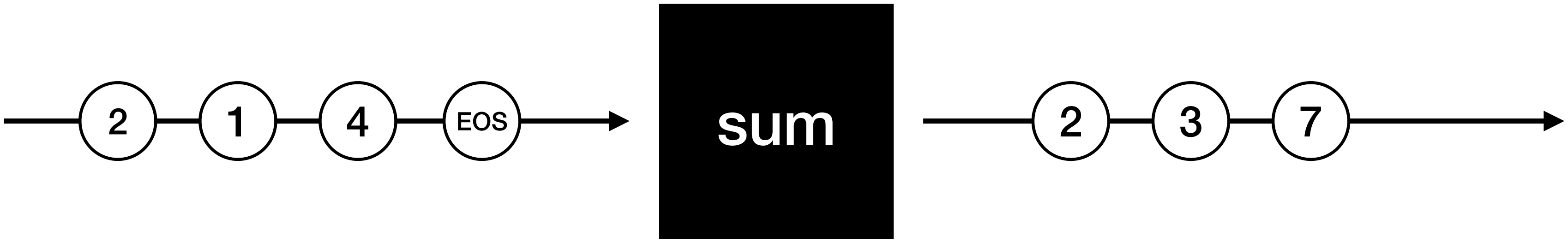
What is Stream Processing?



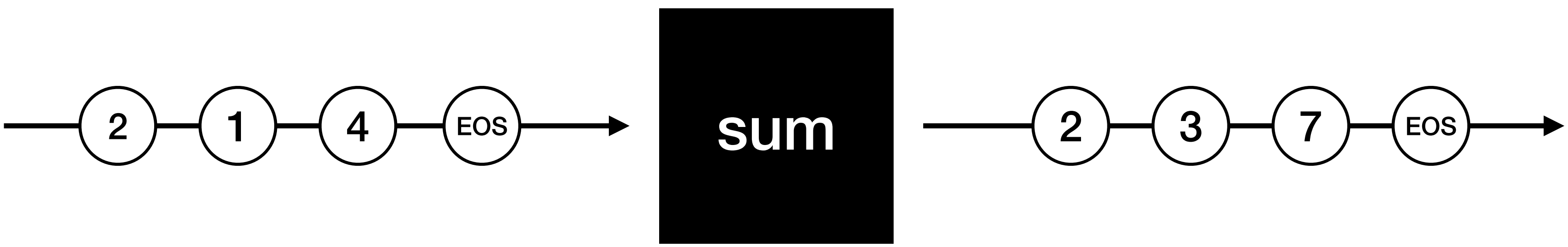
What is Stream Processing?



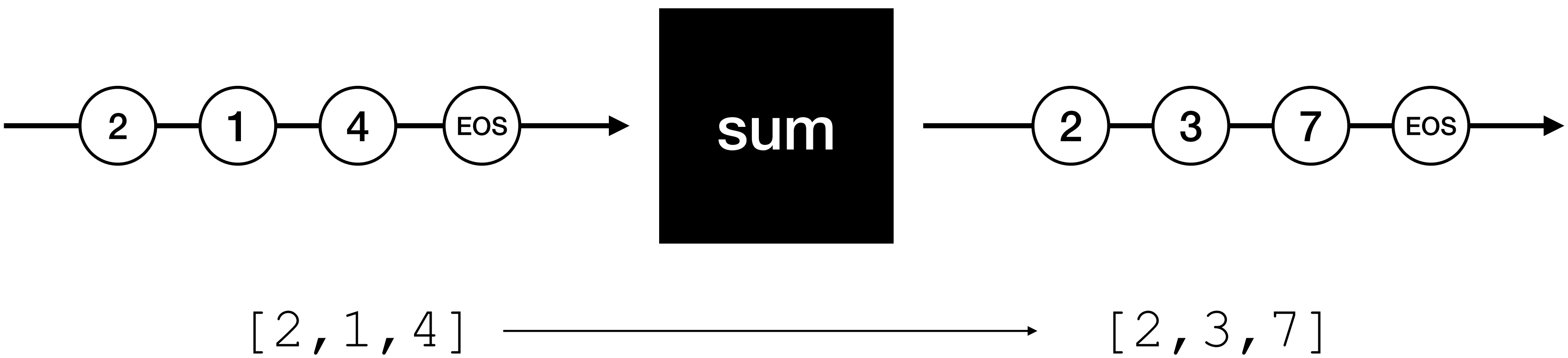
What is Stream Processing?



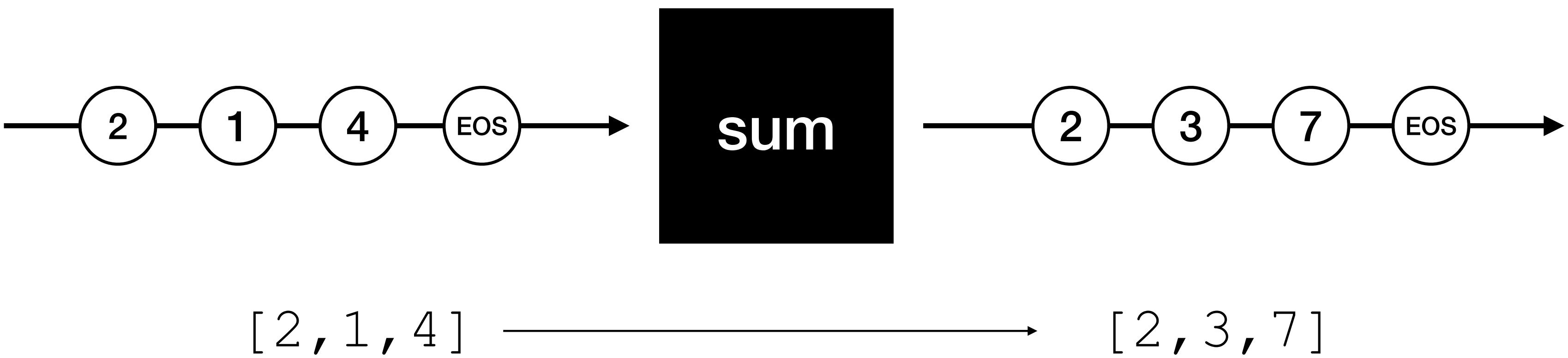
What is Stream Processing?



What is Stream Processing?

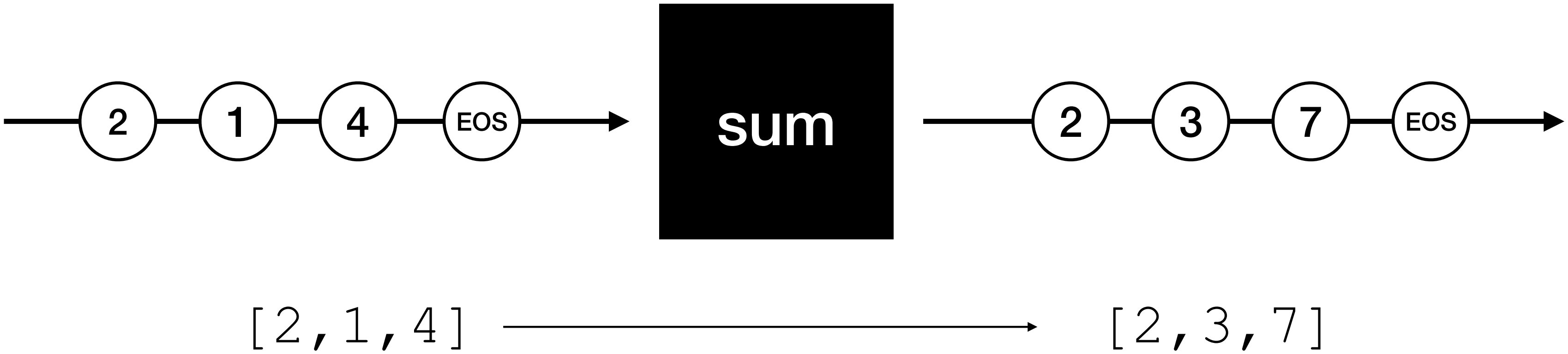


What is Stream Processing?



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

❌ Low-Level

Functional Reactive Programming

```
time -> 'a
```

✅ High-Level ❌ Not Resource-Aware

Streaming eDSL

```
map : ('a -> 'b) -> 'a stream -> 'b stream
filter : ('a -> 'b) -> 'a stream -> 'b stream
window : ('a list -> 'b) -> int -> 'a stream -> 'b stream
fold : ('a -> 'b -> 'b) -> 'b -> 'a stream -> 'b stream
```

✅ Resource Aware

✅ High-Level

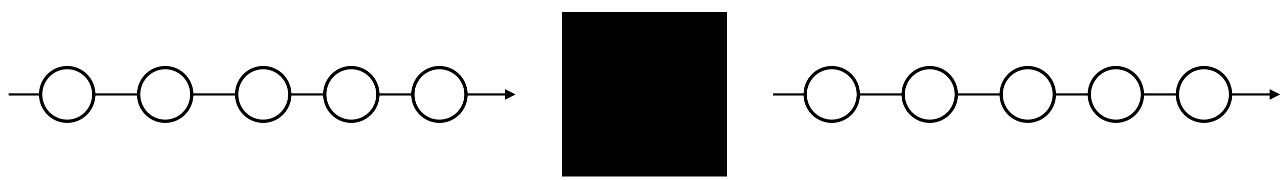
❌ Not Expressive

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)
```

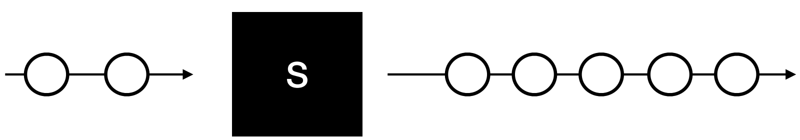
… and **run** it as a streaming program?



delta

Familiar functional syntax…

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



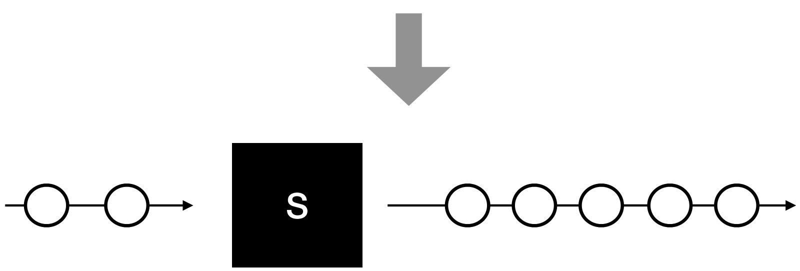
… with streaming semantics!

✅ Expressive ✅ High-Level

delta

Familiar functional syntax…

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

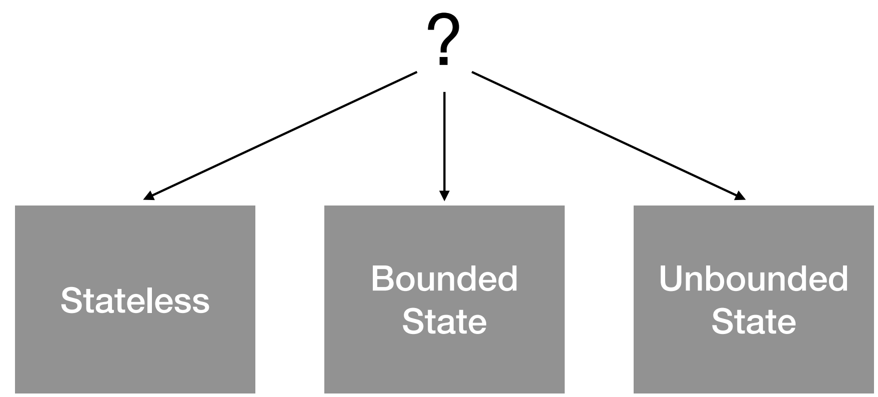


… with streaming semantics!

✅ Expressive ✅ High-Level

And static prevention of space leaks

```
fun foo(xs : int list) = …
```



✅ 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)
```

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

`y` arrives before `ys`…

A *Non*-Example, for Inspiration

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

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

`y` arrives before `ys`…

… but is sent after

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

`y` arrives before `ys`…

… but is sent after

We have to save the *entire stream* in 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

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

$$\underbrace{\Gamma}_{\text{Input Types}} \vdash \underbrace{e}_{\text{Stream Program}} : \underbrace{S}_{\text{Output Type}}$$

Typing Judgment

$$\underbrace{\Gamma}_{\text{Input Types}} \vdash \underbrace{e}_{\text{Stream Program}} : \underbrace{S}_{\text{Output Type}}$$

$$\Gamma = \underbrace{(x_0 : S_0)}_{\text{First variable to arrive}}; (x_1 : S_1); \ldots ; \underbrace{(x_n : S_n)}_{\text{Last variable to arrive}}$$

(Questions?)

Variables

$$\frac{}{\Gamma; (x : S)\,; \Gamma' \vdash x : S}$$

Variables

$$\frac{}{\Gamma; (x : S)\,; \Gamma' \vdash x : S}$$

$$\boxed{x}$$

Variables

$$\frac{}{\Gamma; (x : S) ; \Gamma' \vdash x : S}$$

$$\xrightarrow{\Gamma} \qquad \qquad \boxed{x}$$

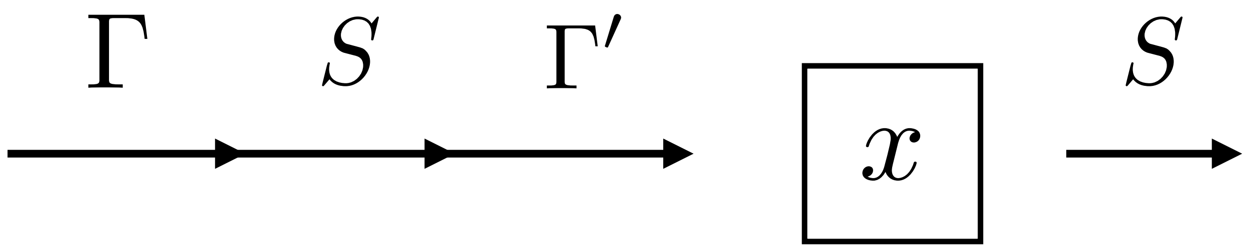
Variables

$$\frac{}{\Gamma; (x : S)\,; \Gamma' \vdash x : S}$$

$$\xrightarrow{\Gamma}\xrightarrow{S} \qquad\qquad \boxed{x} \;\xrightarrow{S}$$

Variables

$$\frac{}{\Gamma; (x : S)\,; \Gamma' \vdash x : S}$$



Nil and Cons

$$\frac{}{\Gamma \vdash \mathtt{nil} : S^{\star}}$$

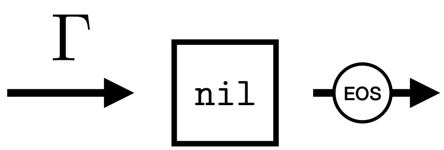
Nil and Cons

$$\frac{}{\Gamma \vdash \texttt{nil} : S^\star}$$

$\boxed{\texttt{nil}}$

Nil and Cons

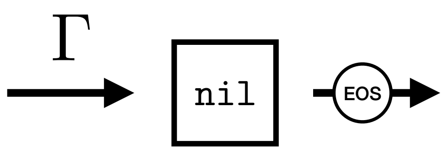
$$\frac{}{\Gamma \vdash \texttt{nil} : S^\star}$$



Nil and Cons

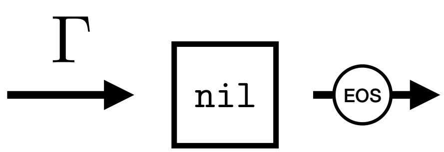
$$\frac{}{\Gamma \vdash \texttt{nil} : S^\star}$$

$$\frac{\Gamma \vdash e : S \qquad \Delta \vdash e' : S^\star}{\Gamma; \Delta \vdash e :: e' : S^\star}$$



Nil and Cons

$$\frac{}{\Gamma \vdash \texttt{nil} : S^\star}$$



Early inputs used for **first output**

Later inputs used for **rest of outputs**

$$\frac{\Gamma \vdash e : S \qquad \Delta \vdash e' : S^\star}{\boxed{\Gamma ; \Delta} \vdash e :: e' : S^\star}$$

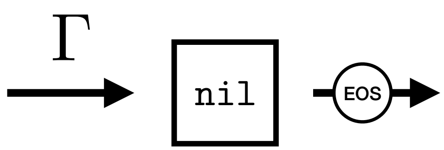
Nil and Cons

Early inputs used for **first output**

Later inputs used for **rest of outputs**

$$\frac{}{\Gamma \vdash \texttt{nil} : S^\star}$$

$$\frac{\Gamma \vdash e : S \qquad \Delta \vdash e' : S^\star}{\boxed{\Gamma ; \Delta} \vdash e :: e' : S^\star}$$



$$\boxed{e :: e'}$$

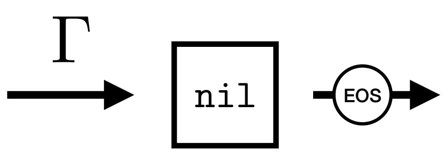
Nil and Cons

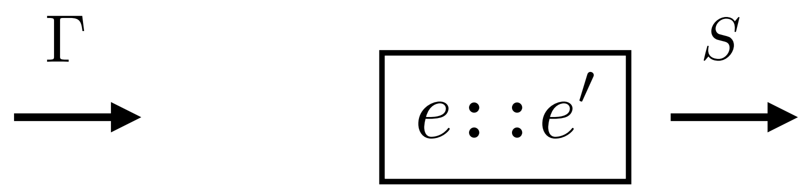
$$\frac{}{\Gamma \vdash \texttt{nil} : S^\star}$$

Early inputs used for **first output**

Later inputs used for **rest of outputs**

$$\frac{\Gamma \vdash e : S \qquad \Delta \vdash e' : S^\star}{\boxed{\Gamma; \Delta} \vdash e :: e' : S^\star}$$





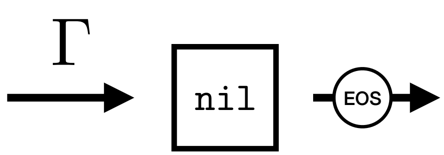
Nil and Cons

$$\frac{}{\Gamma \vdash \texttt{nil} : S^\star}$$

Early inputs used for **first output**

Later inputs used for **rest of outputs**

$$\frac{\Gamma \vdash e : S \qquad \Delta \vdash e' : S^\star}{\boxed{\Gamma ; \Delta} \vdash e :: e' : S^\star}$$



Star-Case

$$\frac{\Gamma; \Gamma' \vdash e : T \qquad \Gamma; (y : S); (ys : S^\star); \Gamma' \vdash e' : T}{\Gamma; (xs : S^\star); \Gamma' \vdash \texttt{case}(xs, e, y.ys.e') : T}$$

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
```

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
```

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

$$\texttt{Int}$$

Singleton Type

Ordered Structure Yields Rich Stream Types

$$S + T$$

Sum Type

$$S \cdot T$$

Concatenation Type

$$\texttt{Int}$$

Singleton Type

$$\mathcal{E}$$

Empty Stream Type

Quiz

What streams do these types describe?

$$\texttt{Int} + \texttt{Int} \cdot \texttt{Int} \qquad \texttt{Int} \cdot (\texttt{Int} + \varepsilon)$$

Technical footnote:

Stream Types **vs.** **Separation Logic**

$$S \cdot T$$

Separation in time

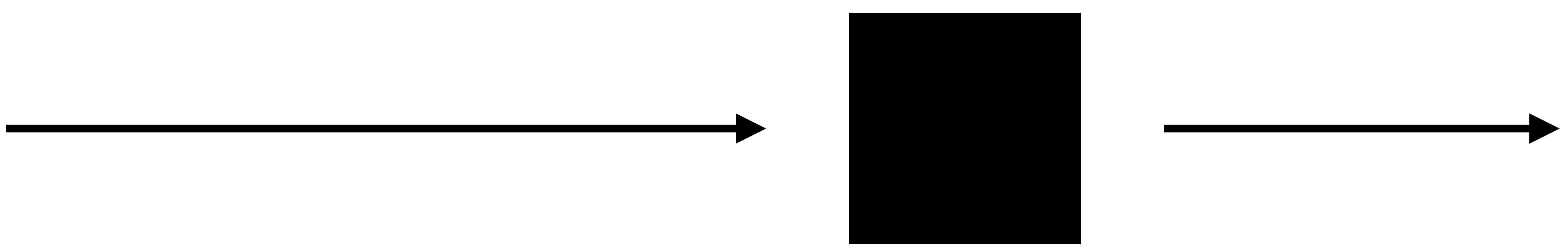
$$S * T$$

Separation in (heap) space

**Rich types
let us build stream programs
*compositionally***

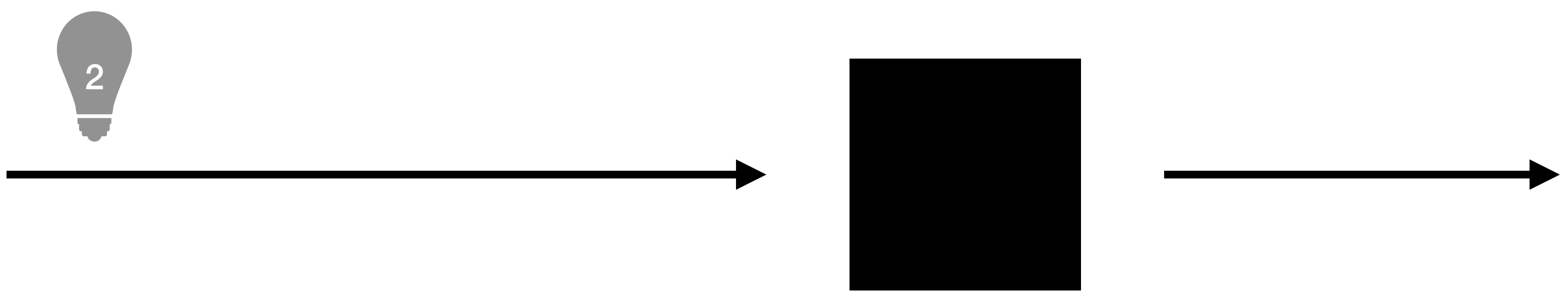
Example

Compute the average of each run above 3



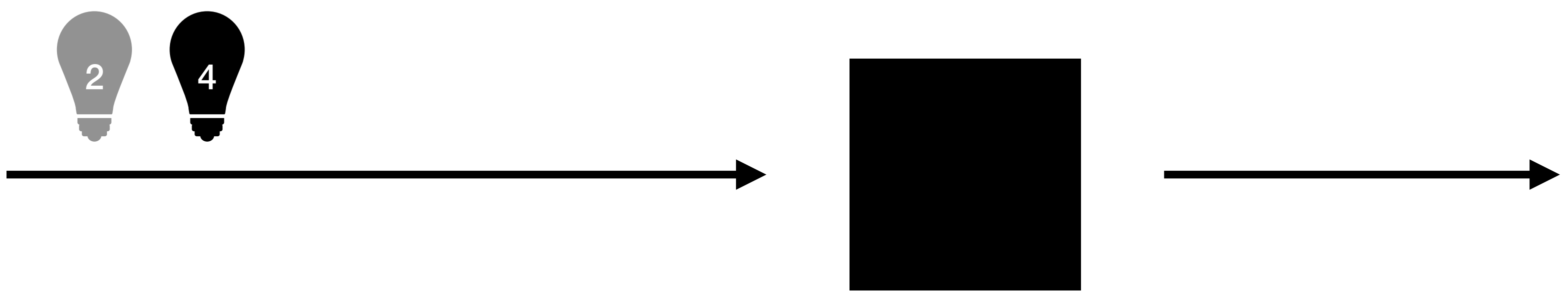
Example

Compute the average of each run above 3



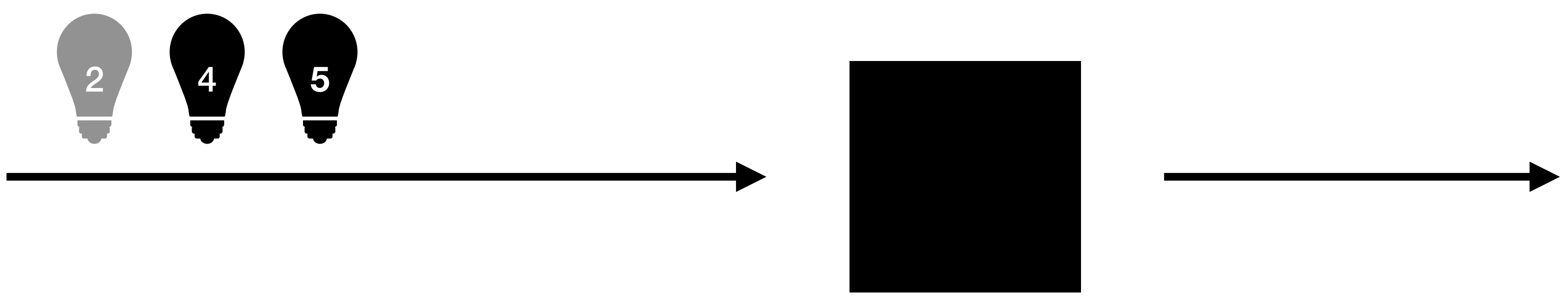
Example

Compute the average of each run above 3



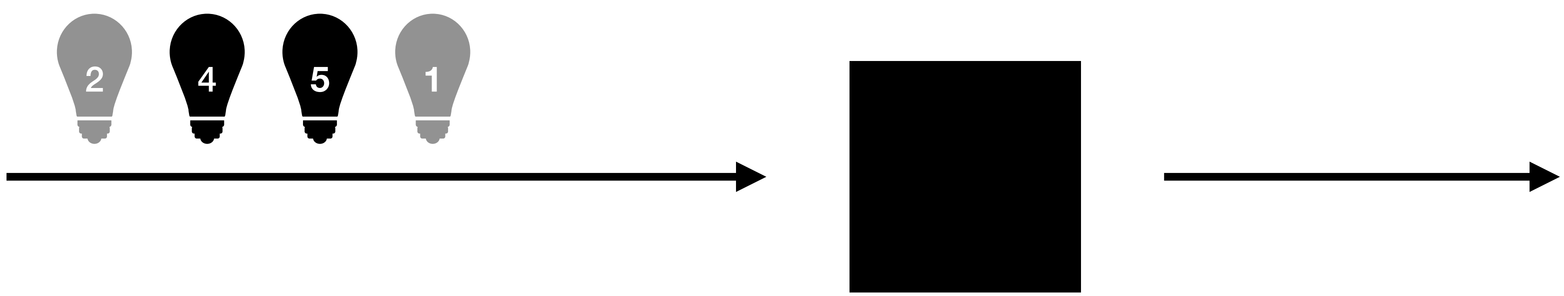
Example

Compute the average of each run above 3



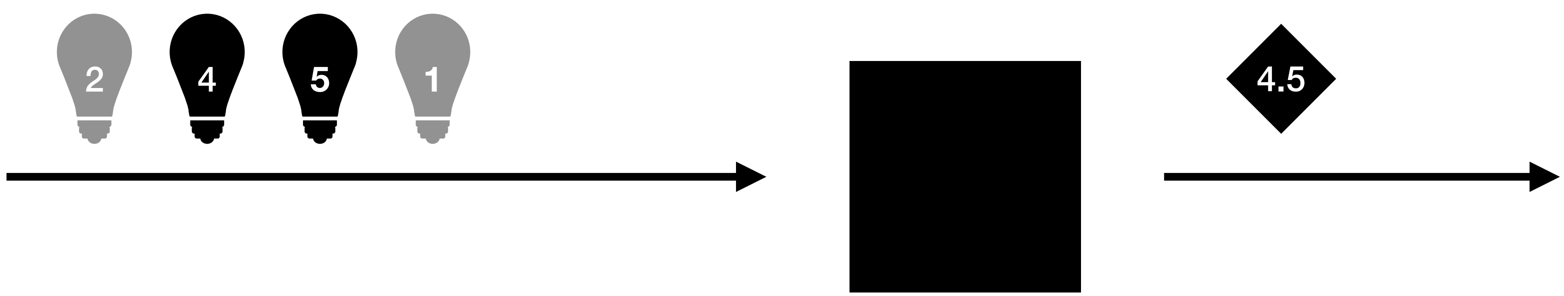
Example

Compute the average of each run above 3



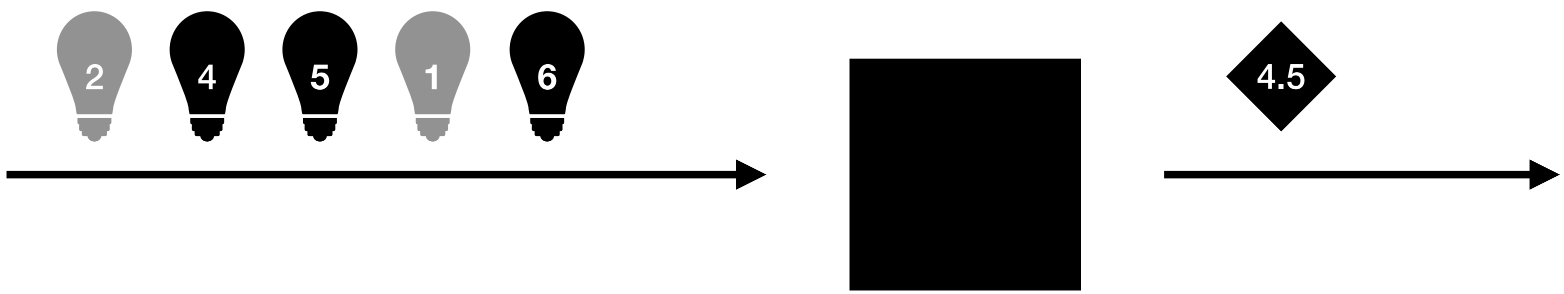
Example

Compute the average of each run above 3



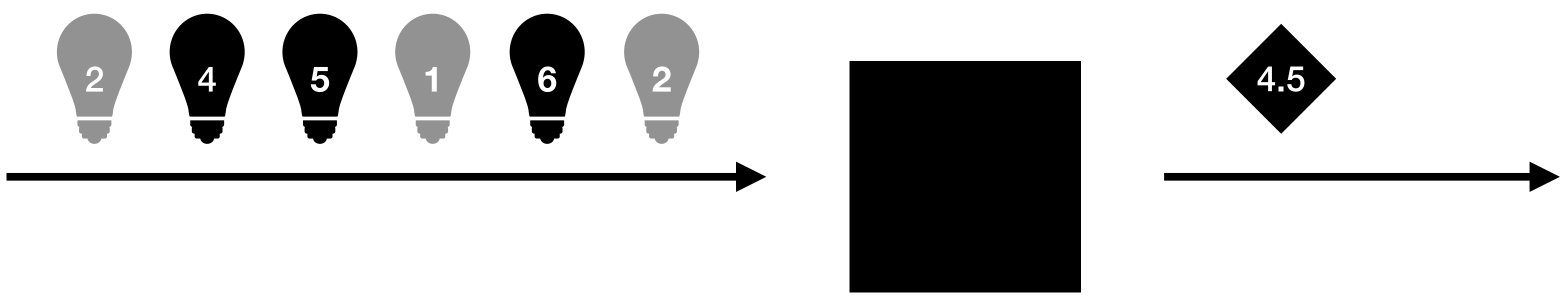
Example

Compute the average of each run above 3



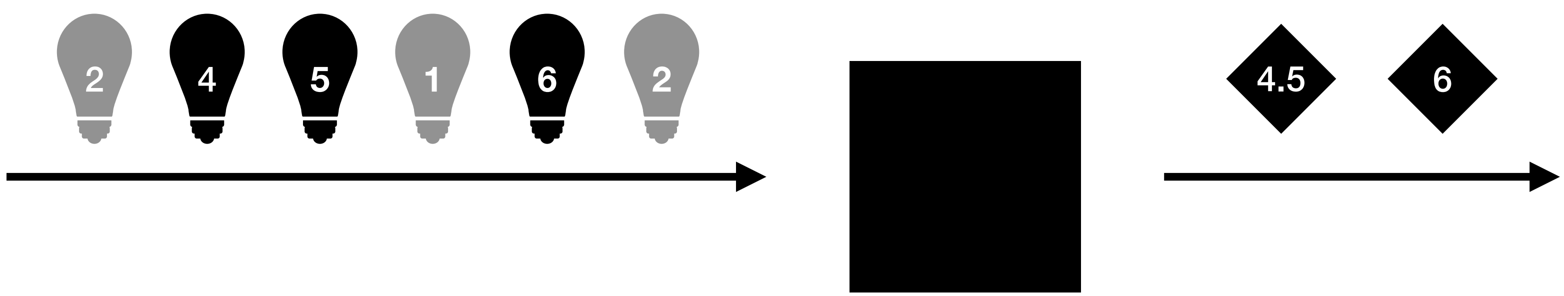
Example

Compute the average of each run above 3



Example

Compute the average of each run above 3



In Flink:

“Compute averages of runs above 3”

In Flink:

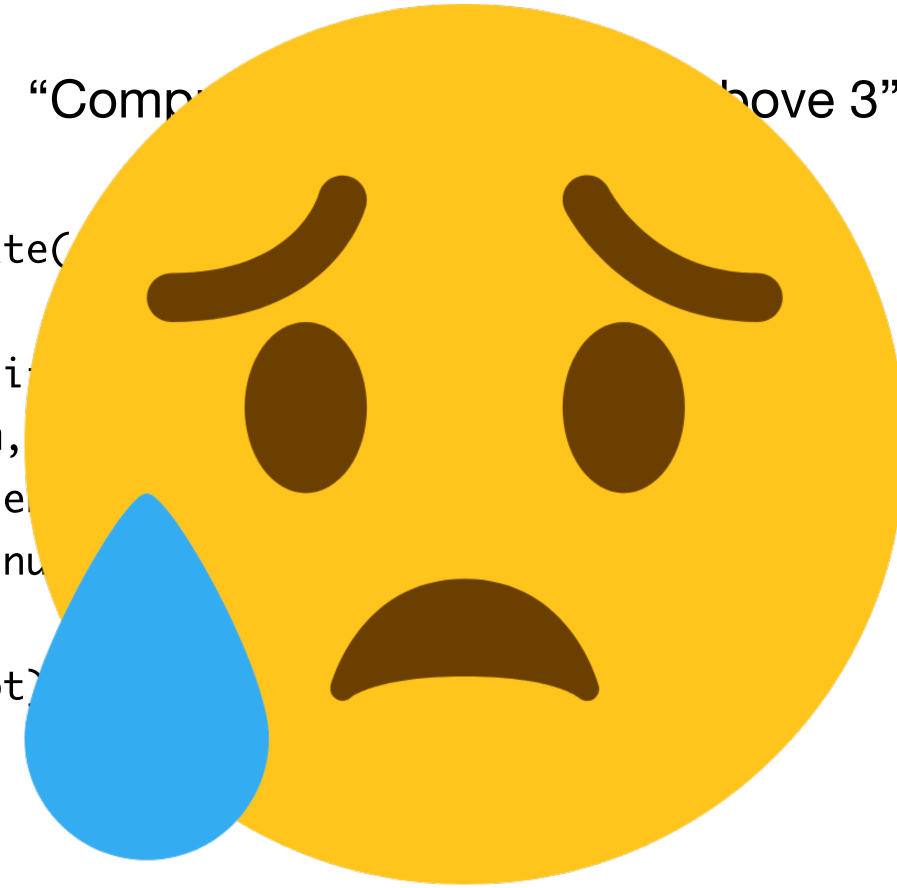
"Compute averages of runs above 3"

```
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:

“Comp … bove 3”

```
xs.flatMapWithState(
  st match {
    case None => i                                    one)
    case Some(num,
      if x > 3 the
        ([],Some(nu
      else
        ([(x + tot)
  }
)
```



In delta:

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

$$\mathtt{Int} \cdot \mathtt{Int}^\star$$

“Nonempty run of Ints”

$$(\mathtt{Int} \cdot \mathtt{Int}^\star)^\star$$

“Stream of nonempty runs of Ints”

Typeful Programming in delta

$$\texttt{Int} \cdot \texttt{Int}^{\star} \xrightarrow{\texttt{avgRun}} \texttt{Int}$$

$$\begin{tikzcd} \texttt{Int}^{\star} \arrow[r, "\texttt{parseRuns}"] \arrow[rr, bend right, dashed, "\texttt{avgAbove3}"'] & (\texttt{Int} \cdot \texttt{Int}^{\star})^{\star} \arrow[r, "\texttt{map(avgRun)}"] & \texttt{Int}^{\star} \end{tikzcd}$$

Typeful Programming in delta

```
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))
```

Typeful Programming in delta

```
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?

$$S_1 \| S_2 \| S_3 \longrightarrow \blacksquare \longrightarrow T_1 \| T_2$$

Multiple inputs are products!

“Parallel Substreams” Type

$$S \| T$$

"A stream of type S interleaved with a stream of type T”

(with elements tagged to indicate their source)

Careful…

Parallel inputs risk **nondeterminism!**

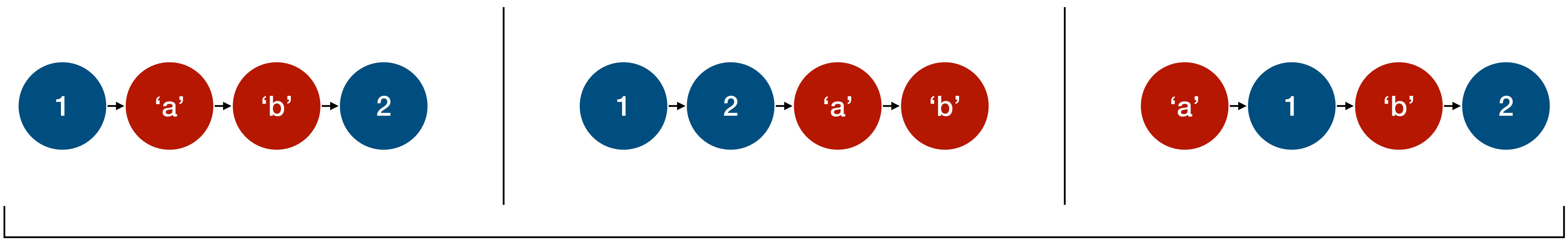
Problem: Multiple Equivalent Interleavings

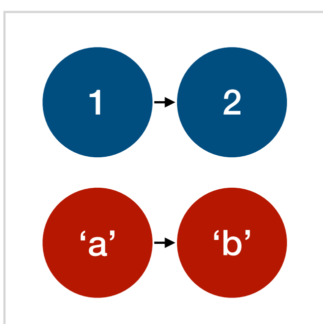
$$\texttt{Int}^\star \,\|\, \texttt{Char}^\star$$

1 → ‘a’ → ‘b’ → 2 | 1 → 2 → ‘a’ → ‘b’ | ‘a’ → 1 → ‘b’ → 2

Problem: Multiple Equivalent Interleavings

$$\mathtt{Int}^{\star} \parallel \mathtt{Char}^{\star}$$





Streams with parallel are actually *partially ordered*

We <u>don't</u> want to be able to write this:

```
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

$$\frac{\Gamma \vdash e : S \qquad \Gamma \vdash e' : T}{\Gamma \vdash (e, e') : S \| T}$$

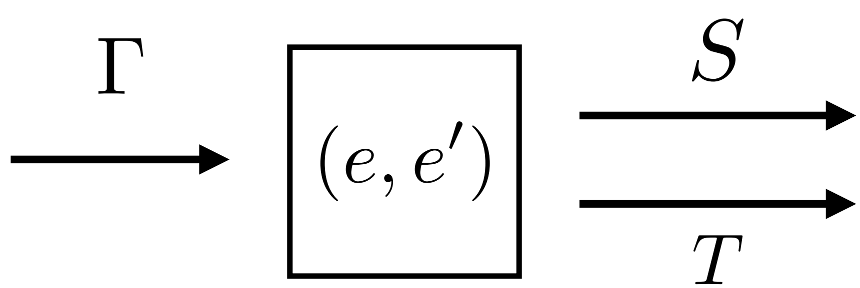
Parallel Right Rule

$$\frac{\Gamma \vdash e : S \qquad \Gamma \vdash e' : T}{\Gamma \vdash (e, e') : S \| T}$$

$$\boxed{(e, e')}$$

Parallel Right Rule

$$\frac{\Gamma \vdash e : S \qquad \Gamma \vdash e' : T}{\Gamma \vdash (e, e') : S \| T}$$



Parallel Left Rule, Take 1

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

Parallel Left Rule, Take 1

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

$(x : S); (y : T)$

Parallel Left Rule, Take 1

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

$(x : S) \cancel{,} (y : T)$

Parallel Left Rule, Take 1

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

$(x : S)\cancel{,}(y : T)$ $\qquad\qquad$ $(y : T); (x : S)$

Parallel Left Rule, Take 1

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

$(x : S)\cancel{,}(y : T)$

$(y : T)\cancel{,}(x : S)$

Parallel Left Rule, Take 1

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

$$(x : S) \cancel{,} (y : T) \qquad (x : S), (y : T) \qquad (y : T) \cancel{,} (x : S)$$

“Bunched” Contexts

$$\Gamma ::= \cdot \mid x : S \mid \Gamma, \Gamma \mid \Gamma; \Gamma$$

“Bunched” Contexts

$$\Gamma ::= \cdot \mid x : S \mid \Gamma, \Gamma \mid \Gamma ; \Gamma$$

Γ, Δ	$\Gamma; \Delta$
Unordered	Ordered

“Bunched” Contexts

$$\Gamma ::= \cdot \mid x : S \mid \Gamma, \Gamma \mid \Gamma; \Gamma$$

Γ, Δ	$\Gamma; \Delta$
Unordered	Ordered
Corresponds to parallel	Corresponds to concat

Unordered data can't be used in an ordered way

$$\frac{\vdots}{(x : S)\,,(y : T) \vdash (x; y) : S \cdot T}$$

Formally: Cat-R rule
requires semicolon context

Unordered data can't be used in an ordered way

$$\frac{\vdots}{(x : S)\,,\,(y : T) \not\vdash (x; y) : S \cdot T}$$

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

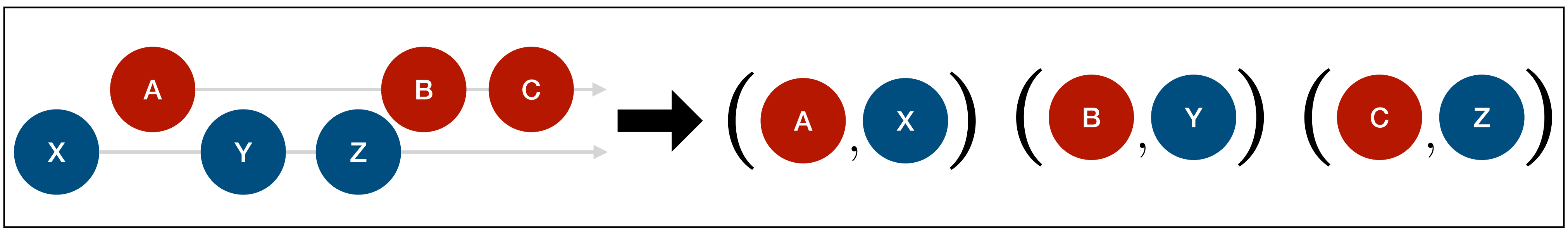
Looks like **list** partition, runs like **stream** partition!

```
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

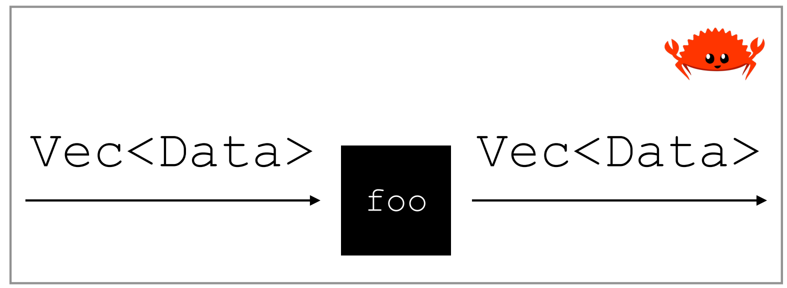
Single-node
Haskell interpreter

Soon

Compiler to Rust-based
runtime above Hydro
dataflow engine

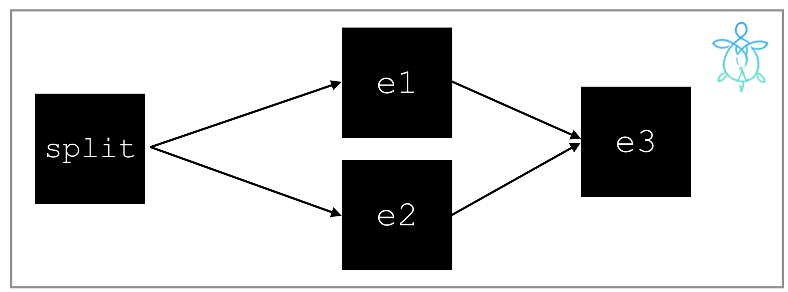
Single-Node Compiler

```
fun foo(x : s) : t = …
```



Multi-Node Compiler

```
fun foo(x : s) : t =
   let (y,z) = (e1,e2)
   in e3
```



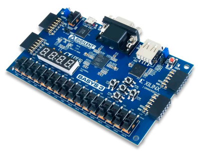
Wrapping up…

Future Directions

Compile Targets

Hydro

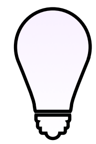
Distributed Systems



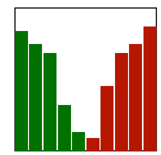
Hardware

Streaming DBs

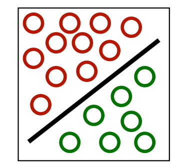
Case Studies



IoT and Edge



Financial Data



ML Streaming

Theory

$[\![\cdot]\!]$

Denotational Semantics

`'a signal`

FRP Connections

$\approx$

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?