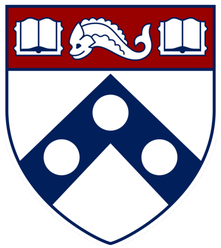



The Essence of Structured Streaming Computation

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

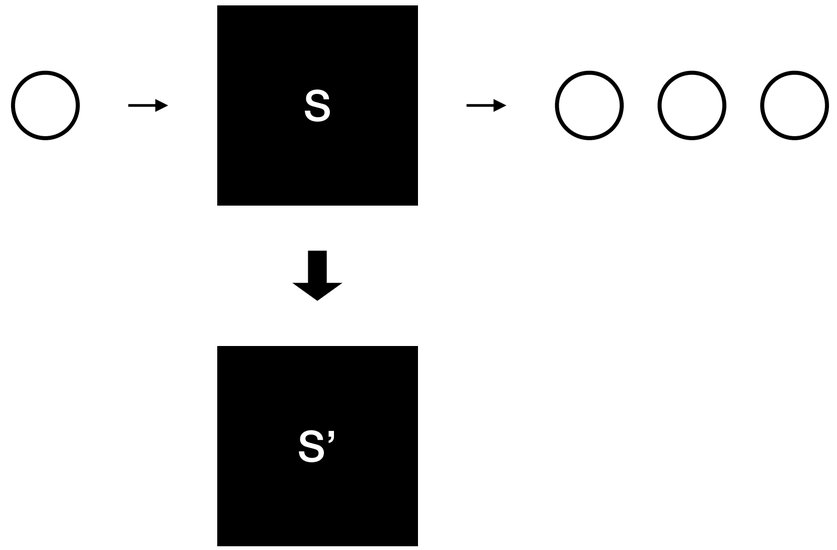
Types are the essence of Structured Streaming

Picking the right types for streams leads you directly to idiomatic stream programming

Let’s define some terms.

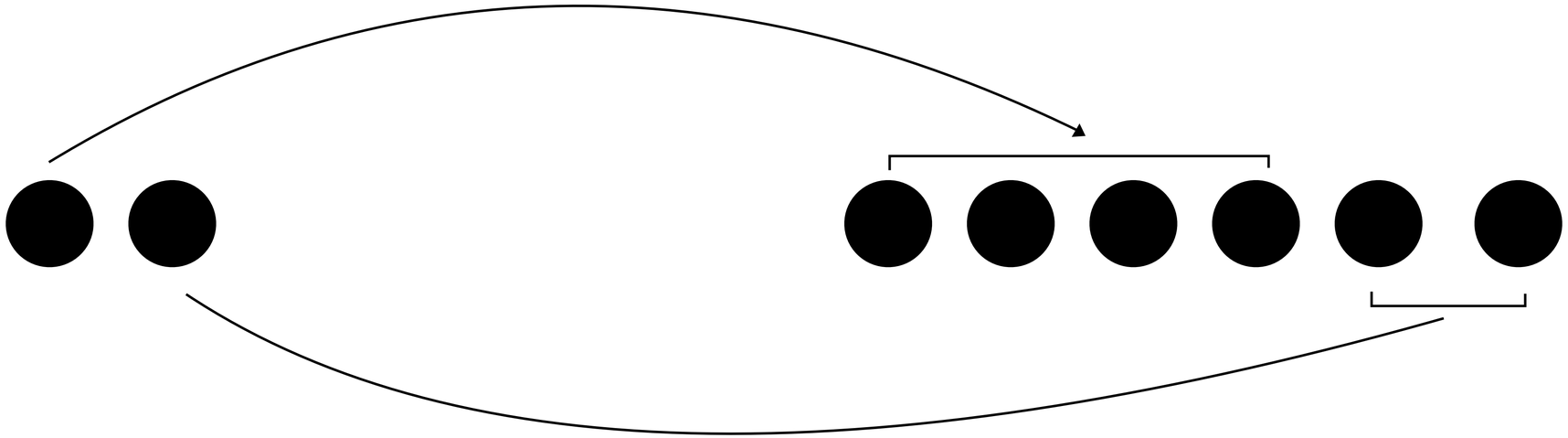
1. What is Streaming Computation?
2. What is this thing I'm calling *structured streaming*?
3. Our Types For Streams
4. The Stream-Typed Core Calculus

What is Streaming Computation?




```
f : state -> input -> state * (list output)
```

Programming Model of Streaming Computation



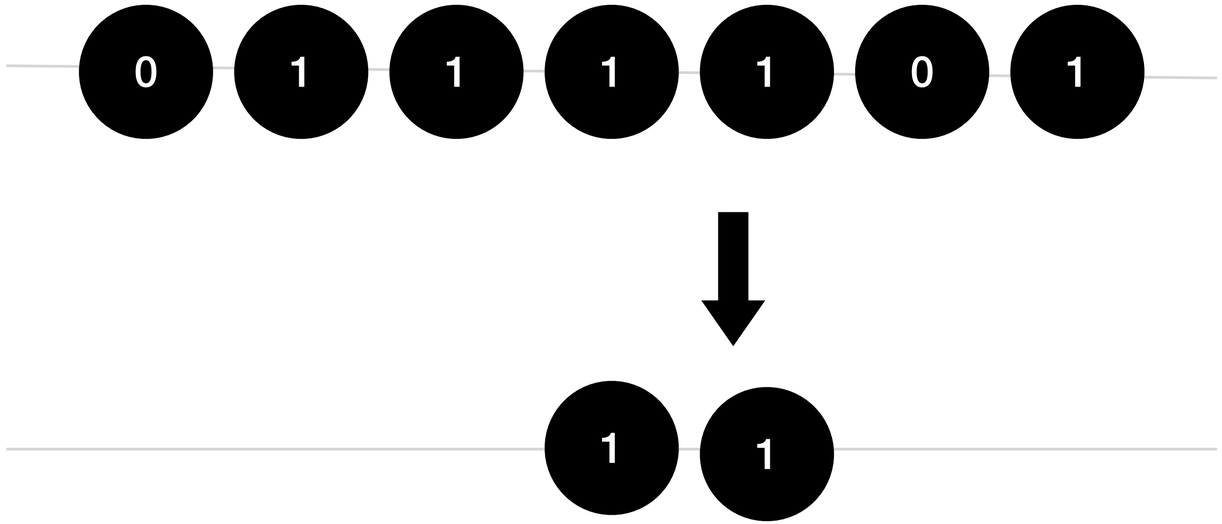
Transform streams, event by event

Example: Streaming Data Analytics

Example: Streaming Data Analytics

Motion Sensor

Alert with 1 if motion is detected in three subsequent periods




```
handle(bool x){
  if(x && prev1 && prev2) {
    out(1);
  }
  prev2 = prev1;
  prev1 = x;
}
```

Manually-Written State Machine!

Example: ML Model Training

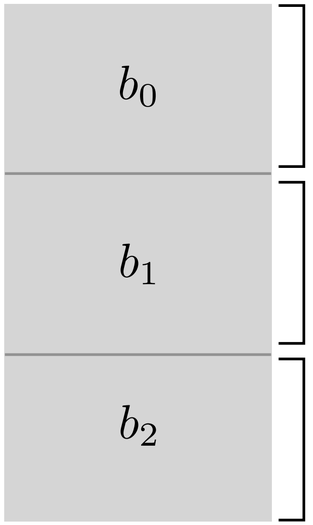
Training data, points at a time

Trained weights

Example: ML Model Training

Minibatch Gradient Descent

Another Manually-Written State Machine!



$$w_1 = w_0 - \frac{\alpha}{k} \sum_{x \in b_0} \nabla L(x, w_0)$$

$$w_2 = w_1 - \frac{\alpha}{k} \sum_{x \in b_1} \nabla L(x, w_1)$$

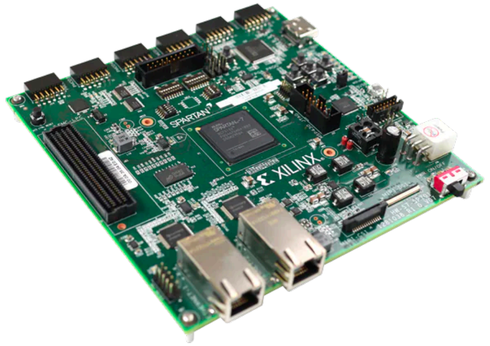
$$w_3 = w_2 - \frac{\alpha}{k} \sum_{x \in b_2} \nabla L(x, w_2)$$

```
handle(vec x){
    loss += gradloss(x,w)
    n += 1
    if(n == k){
        w -= (alpha / k) * loss
        loss,n = 0
    }
}
```

1. Accumulate the loss
2. Update the weights when we've seen k vectors

Example: Networking Hardware

Example: Networking Hardware

01000101010110… →  → 1100111101001…

Extremely Manually-written state machine!

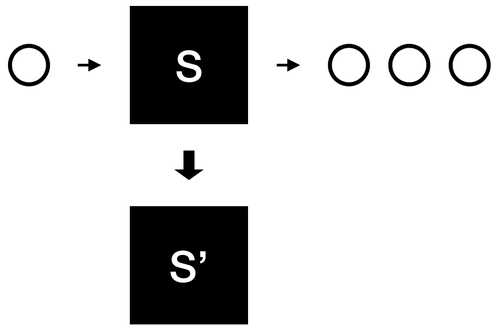
In Summary: Powerful but Challenging!

What is *Structured* Streaming Computation?

Richer Programming Model

```
stream a -> stream b
```

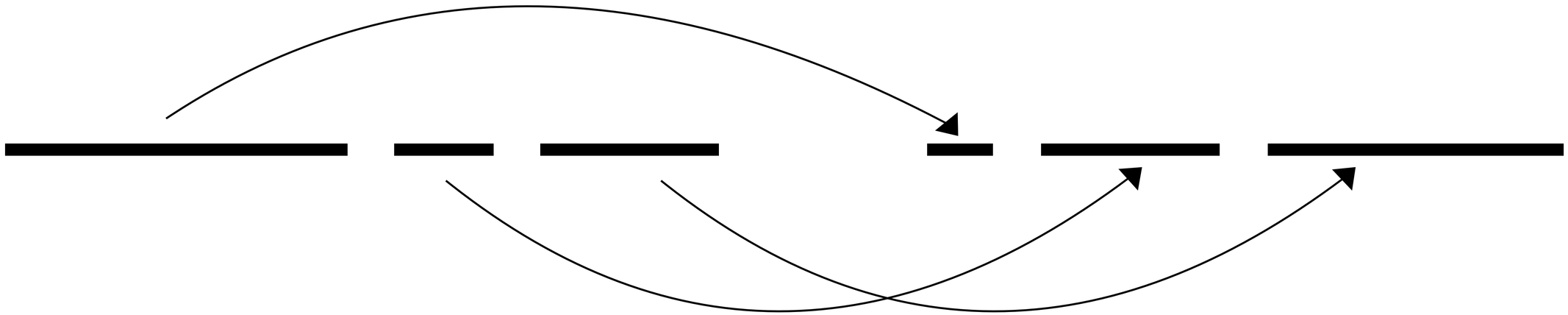
map, filter, reduce, *windows*, and *structured types*



Computational model remains the same

Programming Model of Structured Streaming

Transforming whole extents of a stream at a time



Extents themselves may have variable size, or be composed of structured types

Example: Structured Data Analytics


```
map : (a -> b) -> stream a -> stream b

reduce : (a -> b -> b) -> b -> stream a -> stream b

slidingWindow : int -> (list a -> b) -> stream a -> stream b
```

Example: Structured Streaming Data Analytics

Motion Sensor

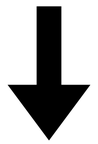
```
handle(bool x){
  if(x && prev1 && prev2) {
    out(1);
  }
  prev2 = prev1;
  prev1 = x;
}
```

```
input.slidingWindow(3,all)
```

Example: Structured ML Model Training

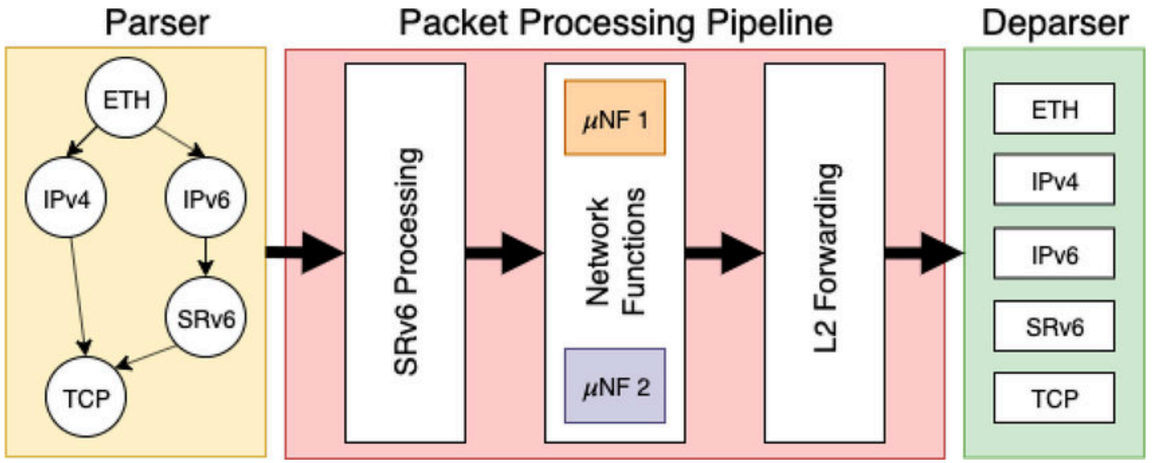
Minibatch Gradient Descent

```
handle(vec x){
    loss += gradloss(x,w)
    n += 1
    if(n == k){
        w -= alpha * (loss / k)
        loss,n = 0
    }
}
```



```
training.map(gradloss).tumblingWindow(k,average_and_update)
```

Example: Structured Networking Hardware Streaming



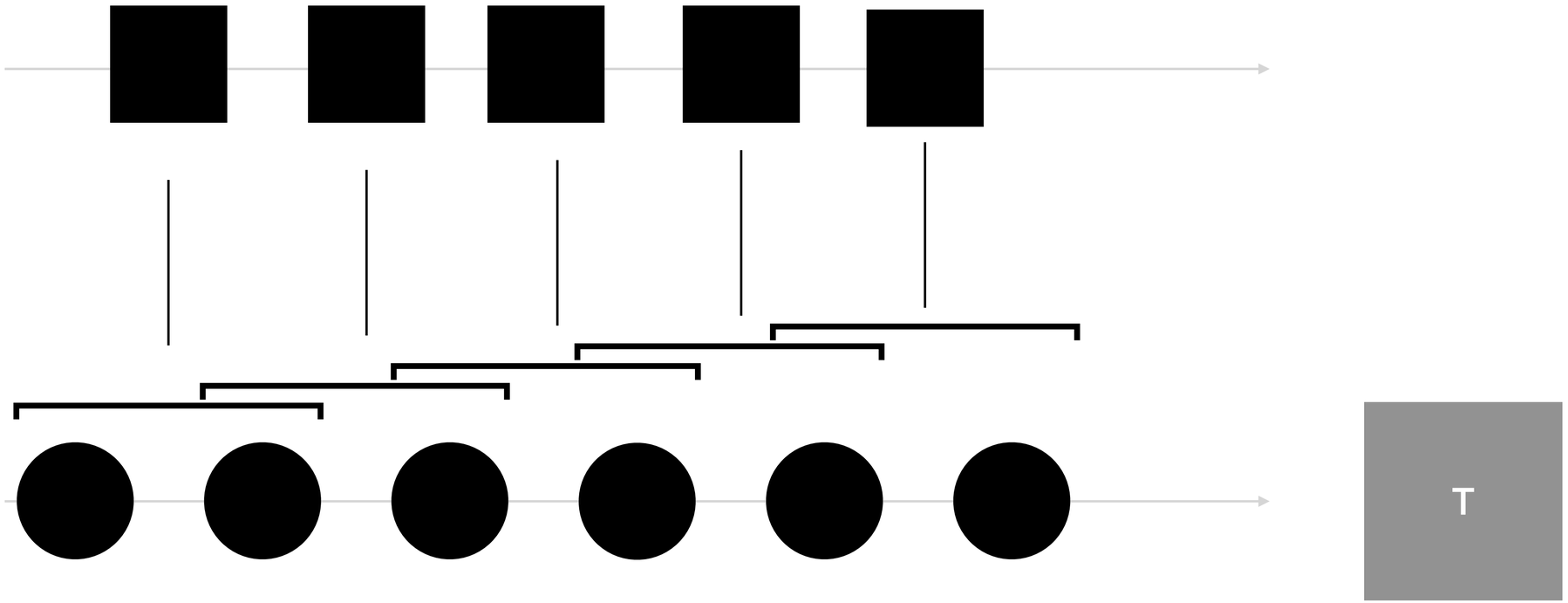

```
parse : stream bit -> stream IPv4

process : stream IPv4 -> stream IPv4

deparse : stream IPv4 -> stream bit
```

Structured Streaming requires ...

Structured Types

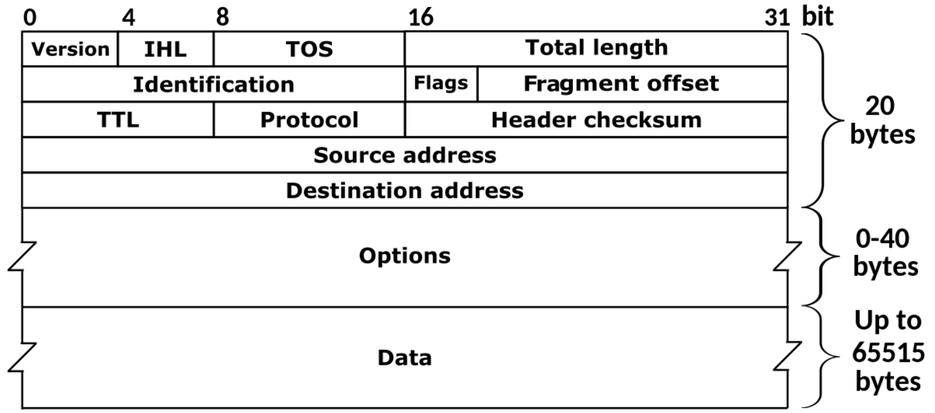
In the rest of this talk, we'll **formalize** my claim by building a *core calculus* of structured streaming

Windows are Streams within Streams




```
type windowed a = stream (stream a)
```

Structured Stream Elements Are Also Streams



$$\texttt{type IPv4} = \texttt{Header} \cdot \texttt{Data}$$

$$\texttt{type Header} = \left(\texttt{version} : \texttt{Bit}^4\right) \cdot \left(\texttt{ihl} : \texttt{Bit}^4\right) \cdot \left(\texttt{dcsp} : \texttt{Bit}^6\right) \ldots$$

$$\texttt{type Data} = \texttt{stream Bit}$$

What Types Are These?

$$s^\star$$

Zero or more streams of type S, one after another

$$s \cdot t$$

A stream of type S, followed by stream of type T

(Ordered Product Type)

$$\mathtt{sing}(\tau)$$

A stream which includes exactly one element of base type

Examples

$(\texttt{sing}\,(\texttt{Int}))^\star$

Traditional Stream of ints, one after another after another.

$((\texttt{sing}\,(\texttt{Int}))^\star)^\star$

Windowed stream of ints

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

Nonempty Stream of Ints

$\texttt{Bit}^\star \cdot \texttt{Bit}^\star$

Stream of bits, followed by another stream of bits

$(\texttt{Bit} \cdot \texttt{Bit})^\star$

Stream of pairs of bits, one after another.

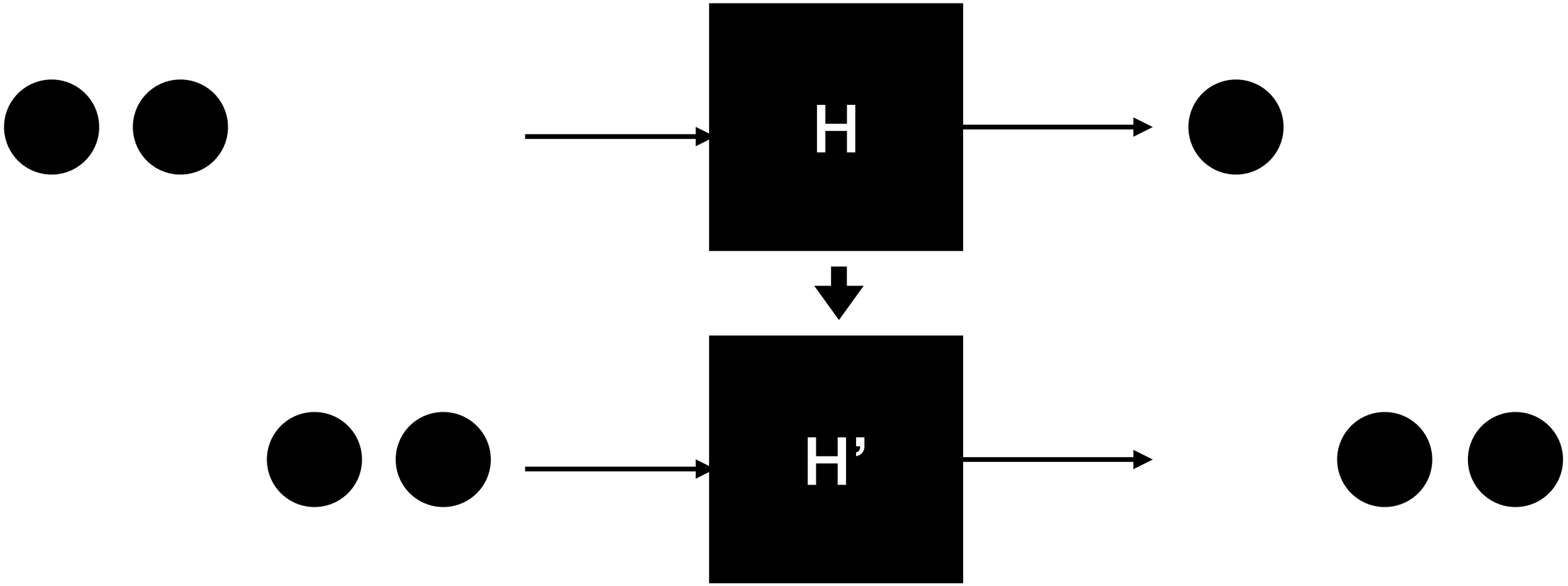
$\texttt{Int}^\star \cdot (\texttt{Char}^\star)^\star \cdot \texttt{Bit}$

Streaming Programs Should Be Typed

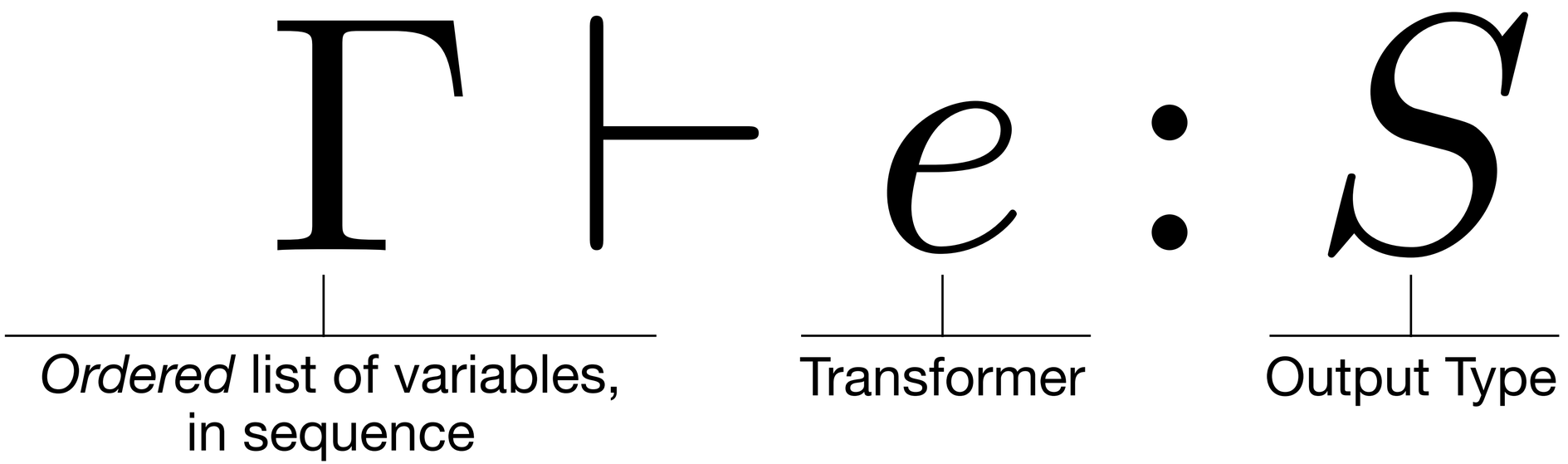
Typed stream transformers must be programmable in a language with a stream-type system

$$x : S \vdash e : T$$

Transformers should have an *incremental* semantics



Typing Judgment



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

Variables range over *structured extents* of the stream

Variable Rule

$$\frac{}{\Gamma; x : s; \Gamma' \vdash x : s}$$

Concat Right Rule, Take 1

$$\frac{\Gamma \vdash e : S \quad \Gamma \vdash e' : T}{\Gamma \vdash (e; e') : S \cdot T}$$

Concat Right Rule, Take 1

$$\frac{\Gamma \vdash e : S \quad \Gamma \vdash e' : T}{\Gamma \vdash (e; e') : S \cdot T}$$

Concat Right Rule

$$\frac{\Gamma \vdash e : S \qquad \Delta \vdash e' : T}{\Gamma; \Delta \vdash (e; e') : S \cdot T}$$

Concat Left Rule

Variables for individual sub-streams

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

Variable for composite of sub-streams

Star Right Rules

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

Star Left Rule

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

Cut Rule

$$\frac{\Delta \vdash e : s \qquad \Gamma; x : s; \Gamma' \vdash e' : t}{\Gamma; \Delta; \Gamma' \vdash \texttt{let}\, x = e\, \texttt{in}\, e' : t}$$

Does this give us the idioms of structured streaming?

Some Programs!

Map

```
map (f : s -> t) (xs : s*) : t* =
 case xs of
  | nil => nil
  | y::ys => (f y) :: map f ys
```

Looks like list map, runs like stream map

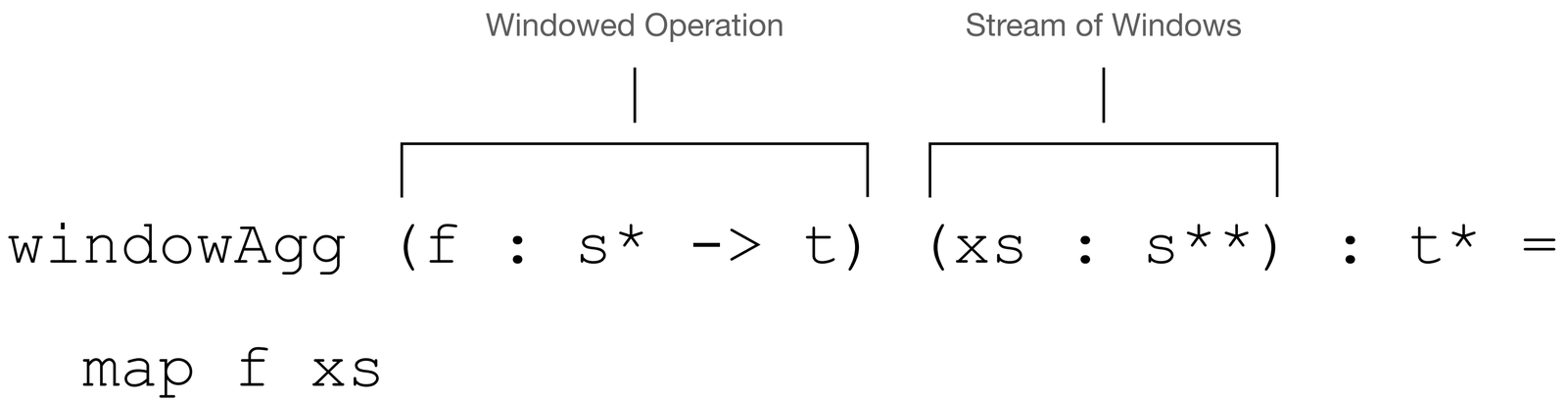
Some Programs!

Filter

```
filter (f : s -> bool) (xs : s*) : s* =
case xs of
  | nil => nil
  | y::ys => if f y then y :: filter ys
             else filter ys
```

Some Programs!

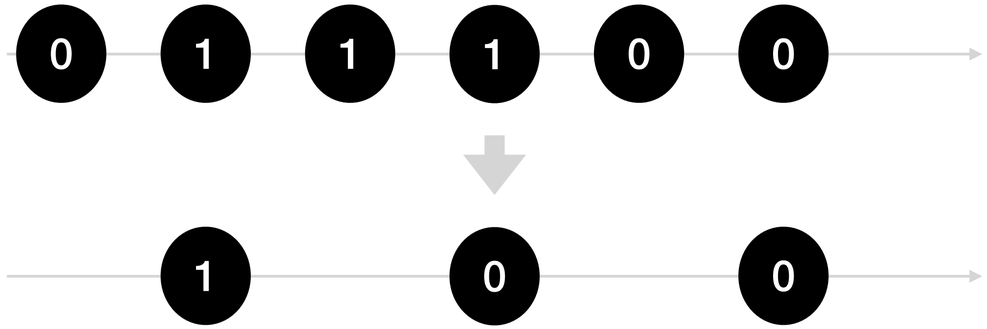
Windowed Aggregation



Windowed aggregation is a map over a stream of windows

Some Programs!

Pairwise Xor of a Stream



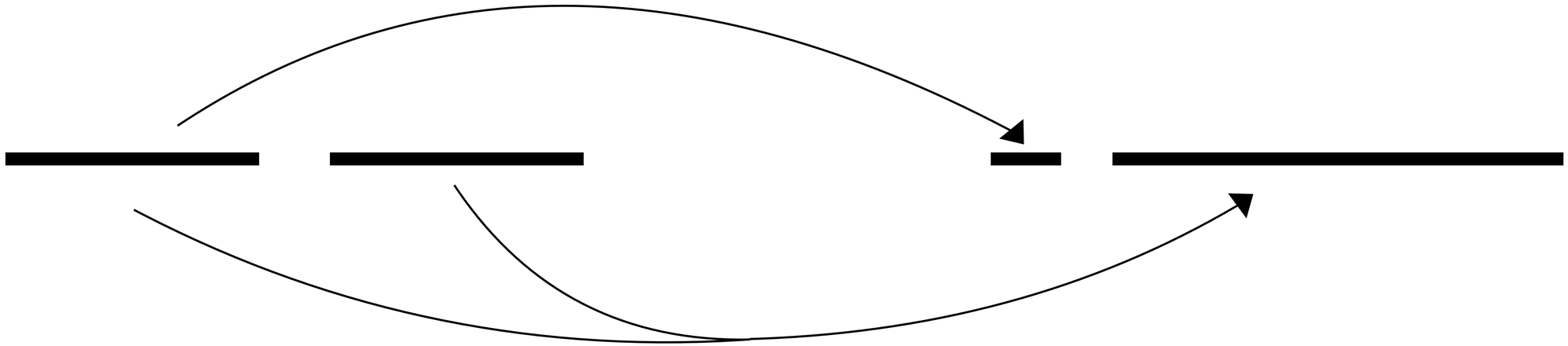

```
parse (xs : bit*) : (bit . bit)* =
case xs of
| nil => nil
| [_] => nil
| b::b'::bs => (b;b') :: parse bs
```

```
xor (x : bit . bit) : bit
 let (a;b) = x in
 if a then !b else b
```

```
pairsxor (xs : bit*) : bit* =
   map xor (parse xs)
```

What About State?

Mental Model of *Stateful* Structured Streaming



Earlier extents of the input can be used to produce later extents of output

State Context

$$\Omega \mid \Gamma \vdash e : S$$

Fully structural context bound to “already-seen data”

Wait Rule

x now in the past

$$\frac{\Omega, x : \langle s \rangle \mid \Gamma \vdash e : t}{\Omega \mid x : s; \Gamma \vdash \texttt{wait}_x\{e\} : t}$$

Stream with substream bound to x

Wait for x then run e

Using State Variables

$$\frac{}{\Omega, x : \langle S \rangle \mid \Gamma \vdash x : S}$$

Examples, With State!

Some Programs!
Window Creation (Sliding)

```
slidingWindow {prev : s*} (xs : s*) : s** =
 case xs of
 | nil => nil
 | y::ys => wait y {
              let next = update prev y in
              next :: slidingWindow next ys
            }
```

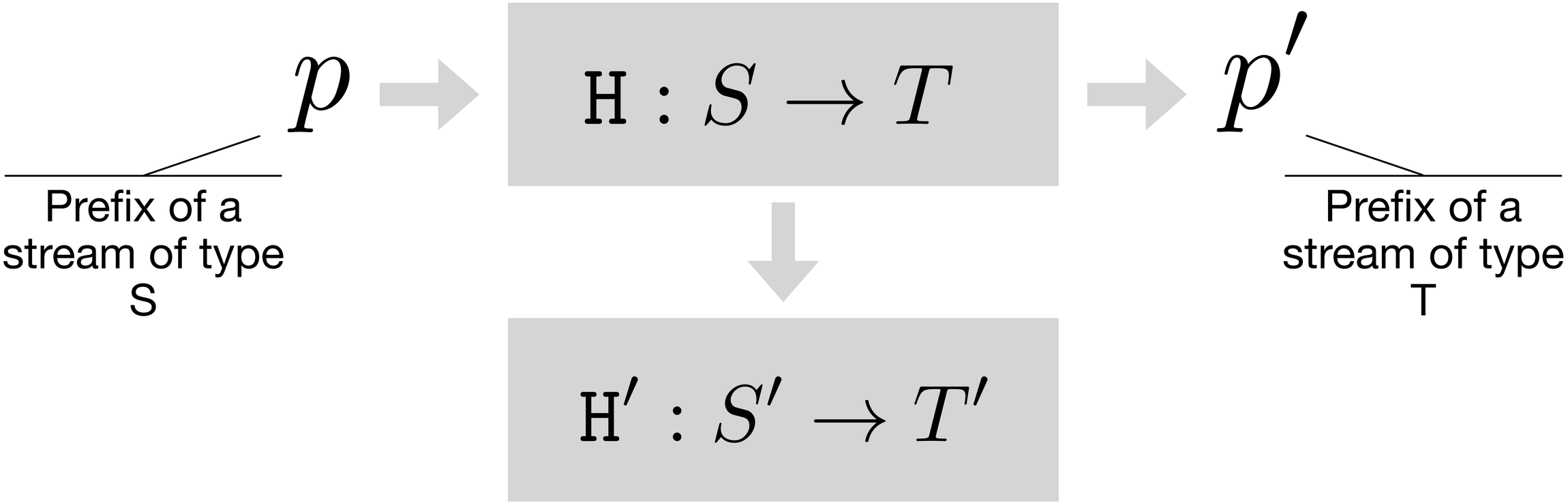
Some Programs!

Window Creation (Tumbling)

```
tumblingWindow {k : int} {acc : s*} (XS : s*) : s** =
  case xs of
  | nil => nil
  | y::ys => if acc.length == k then
                (y :: acc) :: tumblingWindow k [] ys
             else
               wait y {
                 tumblingWindow k (y :: acc) ys
               }
```

Semantics

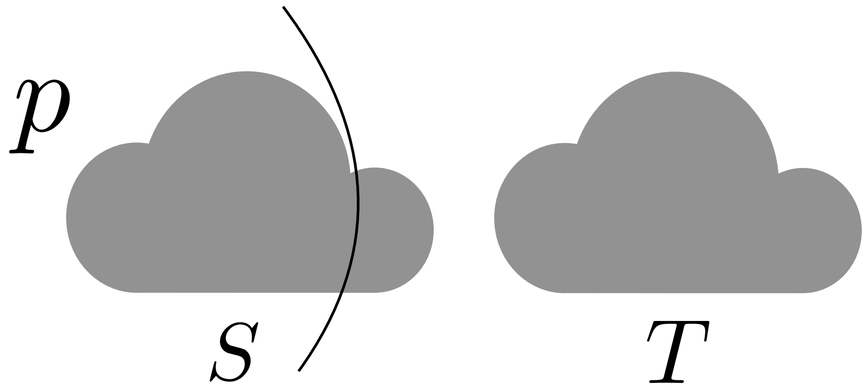
Semantics



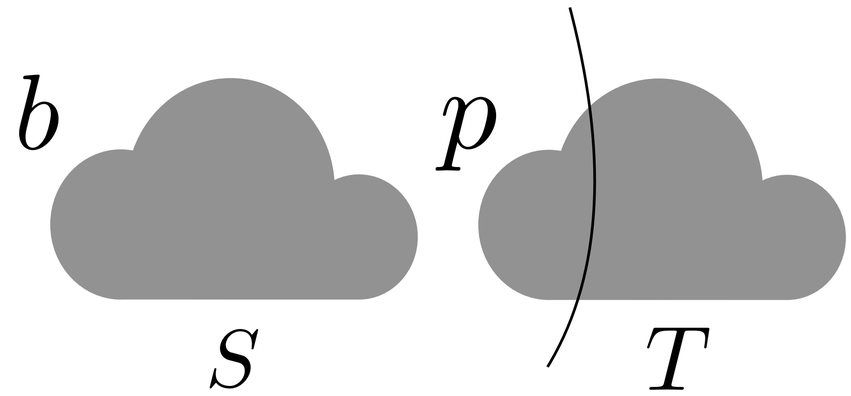
New transformer, accepting the rest of S, and producing the rest of T

What is a Prefix of a Stream of Type S?

Concat



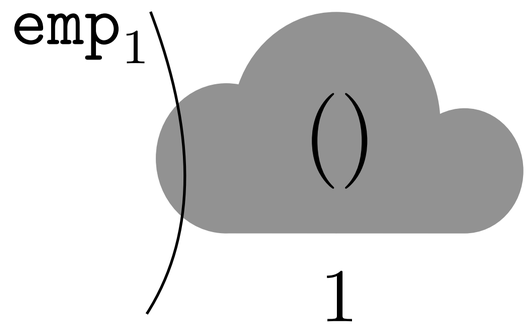
$$\frac{p : \texttt{prefix}(S)}{\texttt{inl}(p) : \texttt{prefix}\,(S \cdot T)}$$



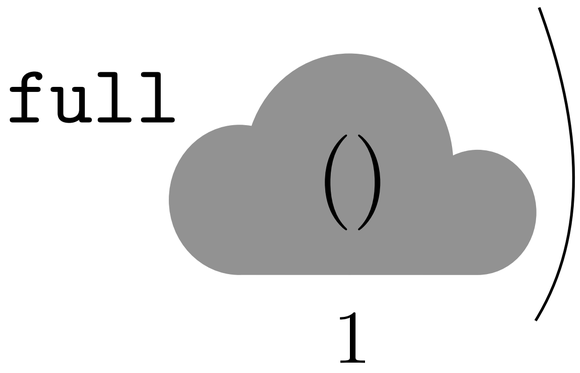
$$\frac{b : \texttt{batch}(S) \quad p : \texttt{prefix}(T)}{\texttt{inr}(b, p) : \texttt{prefix}\,(S \cdot T)}$$

What is a Prefix of a Stream of Type S?

Base Type (1)

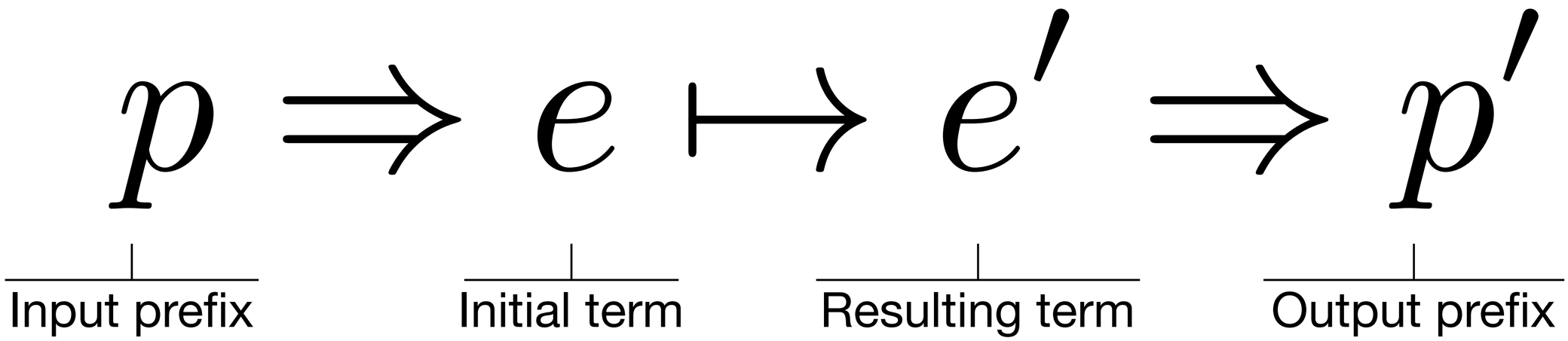


$$\frac{}{\texttt{emp}_1 : \texttt{prefix}(1)}$$



$$\frac{}{\texttt{full} : \texttt{prefix}(1)}$$

Semantics


$p \Rightarrow e \mapsto e' \Rightarrow p'$
Input prefix
Initial term
Resulting term
Output prefix

Preservation

If:

$$\begin{bmatrix} \cdot \mid \Gamma \vdash e : S \\ p : \texttt{prefix}(\Gamma) \\ p \Rightarrow e \mapsto e' \Rightarrow p' \end{bmatrix}$$

Then:

$$\begin{bmatrix} p' : \texttt{prefix}\,(S) \\ \delta_p \Gamma \vdash e' : \delta_{p'} S \end{bmatrix}$$

Progress

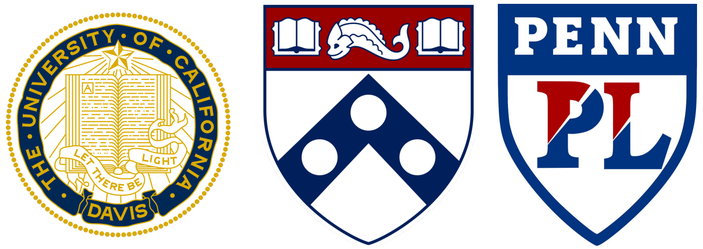
If:

$$\begin{bmatrix} \cdot \mid \Gamma \vdash e : S \\ p : \texttt{prefix}(\Gamma) \end{bmatrix}$$

Then:

$$\exists p, e' .\ p \Rightarrow e \mapsto e' \Rightarrow p'$$

Thank you!



Type System

$$\Gamma \vdash e : S$$

Operational Semantics

$$\eta \Rightarrow e \mapsto e' \Rightarrow p$$

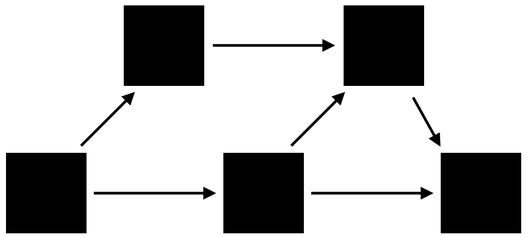
Mechanized Metatheory

$$\frac{\Gamma \vdash e : S \qquad p : \texttt{prefix}(\Gamma)}{\exists p, e' .\ p \Rightarrow e \mapsto e' \Rightarrow p'}$$

Expressiveness

CQL ->
Stream Types

Low-Level Dataflow Model



Interpreter in Haskell

```
{-
G |- e1 : s    D |- e2 : t
-----------------------
G;D |- (e1;e2) : s . t
-}
eval (CatPfxA p) (CatPair e1 e2) = do
  ER {output = p', newExpr = e1'} ← eval p e1
  return $ ER {output = CatPfxA p', newExpr = CatPair e1' e2}

eval (CatPfxB b p) (CatPair e1 e2) = do
  b' ← batchEval b e1
  ER {output = p', newExpr = e2'} ← eval p e2
  return $ ER {output = CatPfxB b' p', newExpr = e2'}
```