Stateful synchronous dataflow language.

Harnessing distributive laws of a comonad over a monad.

Benjamin Lion David Nowak Jean-Pierre Talpin

Motivations

Dataflow language are very popular for high level programming.
Lustre, Esterel, Signal, Circuits, ...

Hybrid language to integrate stateful computations.
Modes, automata, control, ...

Can we define a stateful dataflow language, that merges both?
Consequence for compilation, hands-on memory, ...

What are stateful synchronous dataflow programs?

Non-Stateful Synchronous dataflow programs

```
nats := 0 fby (1 + nats)
```

Input:	`tt`,	`tt`,	`tt`,	`tt`,	`tt`,	`...`
Output:	`0`,	`1`,	`2`,	`3`,	`4`,	`...`

What are stateful synchronous dataflow programs?

Non-Stateful Synchronous dataflow programs

```
nats := 0 fby (1 + nats)
```

Input:	tt,	tt,	tt,	tt,	tt,	...
Output:	0,	1,	2,	3,	4,	...

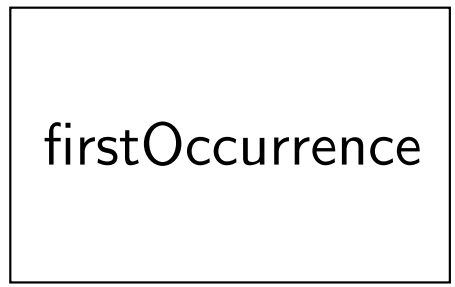
Stateful synchronous dataflow programs

```
nats := let x = (read 0) in (write 0 (1 + x)) ; x
```

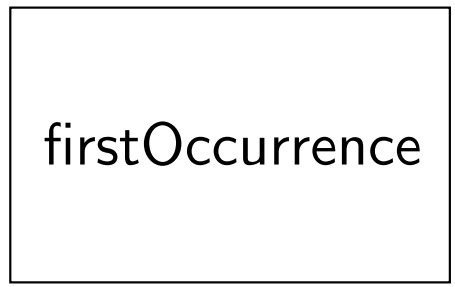
Memory at address 0 initialized to 0.

Input:	tt,	tt,	tt,	tt,	tt,	...
Output:	0,	1,	2,	3,	4,	...

Motivating example: first occurrence

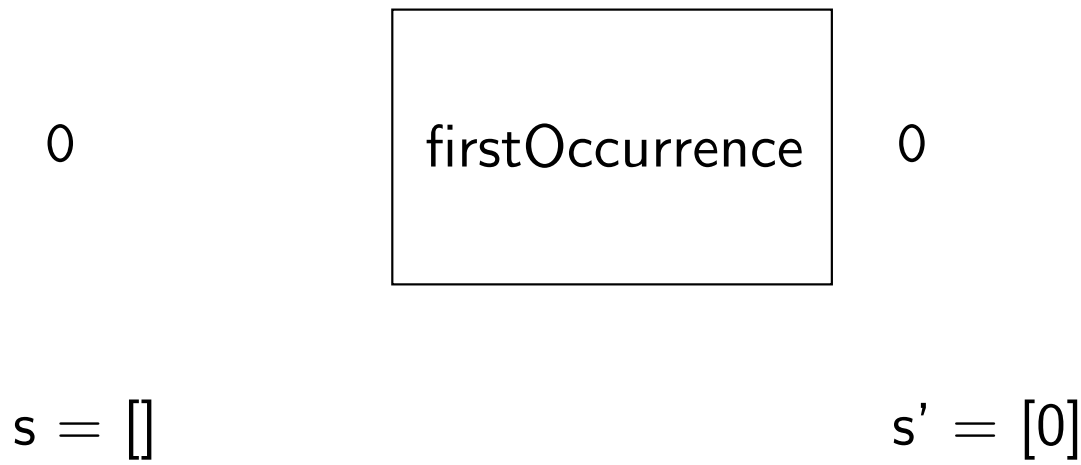


Motivating example: first occurrence

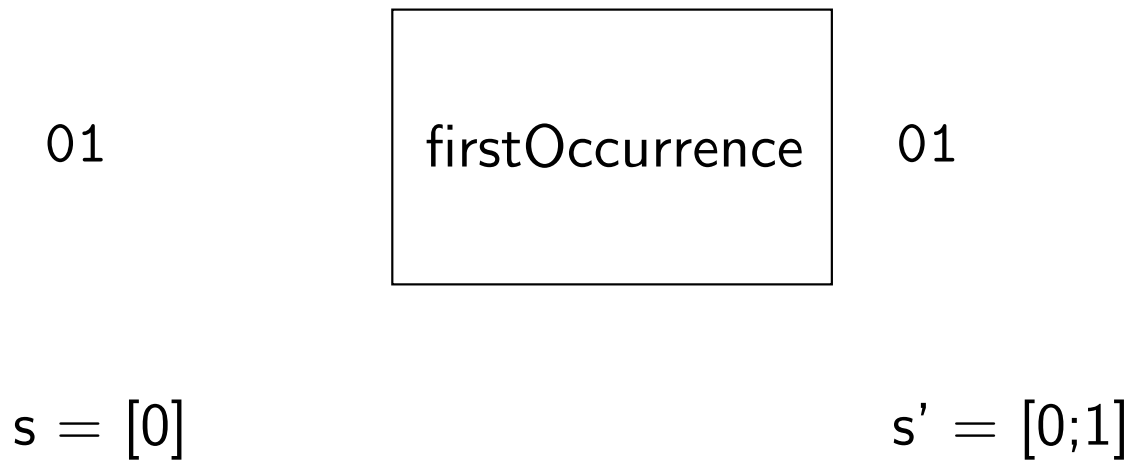


$s = []$

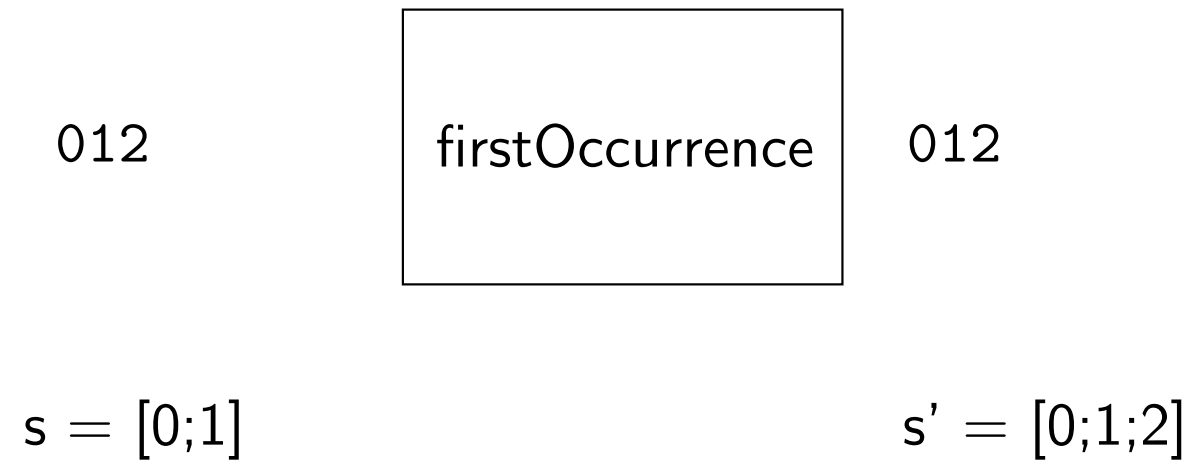
Motivating example: first occurrence



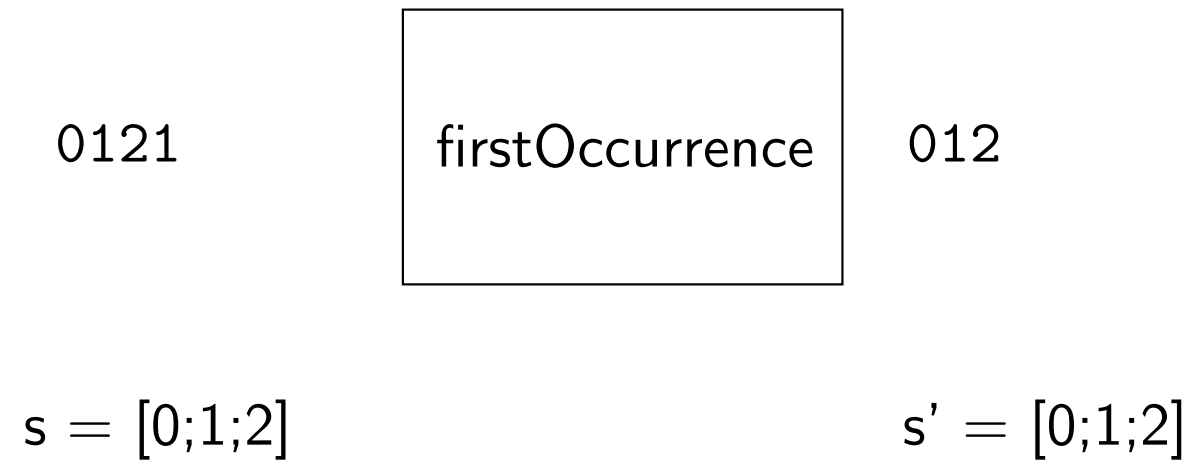
Motivating example: first occurrence



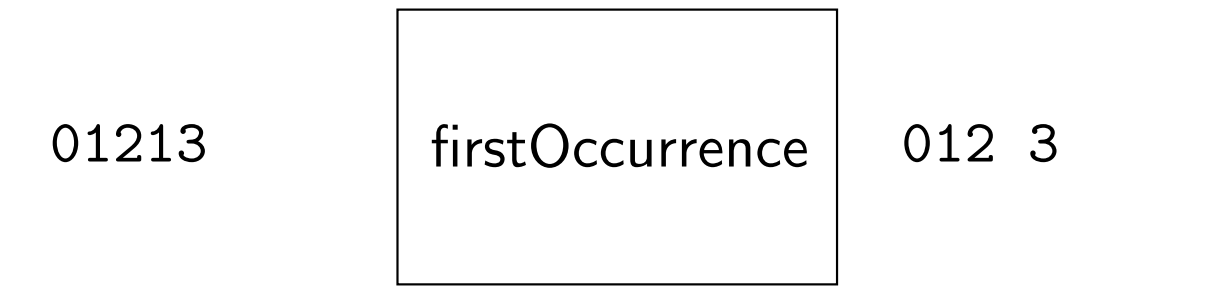
Motivating example: first occurrence



Motivating example: first occurrence



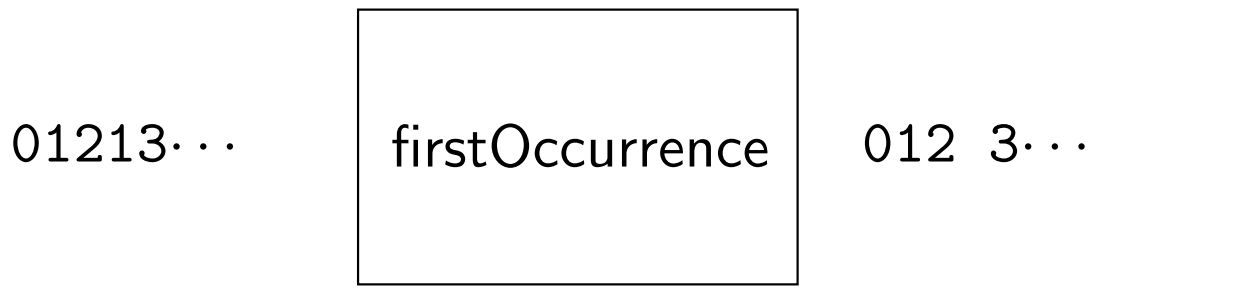
Motivating example: first occurrence



s = [0;1;2]

s' = [0;1;2;3]

Motivating example: first occurrence



s = [0;1;2;3;...]

s' = [0;1;2;3;...]

Semantics

What semantics for such stateful dataflow programs?

Our requirements:

- denotational (make use of Coq for proofs)
- compositional (nicer, and makes proofs easier)

Semantics

What semantics for such stateful dataflow programs?

Our requirements:

- denotational (make use of Coq for proofs)
- compositional (nicer, and makes proofs easier)

Prior investigations of Uustalu and Vene.
Denotational and compositional semantics for Lustre (synchronous dataflow language): dataflow primitives are morphisms in a Kleisli category, i.e.,

$$f : W\ A \to M\ B$$

where W is a comonad and M is a monad.

Dataflow with comonad

Comonad for compositional context:

$$\frac{f : W\ A \to B \qquad g : W\ B \to C}{?? : W\ A \to C}$$

Dataflow with comonad

Comonad for compositional context:

$$\frac{f^\dagger : W\ A \to W\ B \qquad g : W\ B \to C}{g \circ f^\dagger : W\ A \to C}$$

Dataflow with comonad

Comonad for compositional context:

$$\frac{f^\dagger : W\ A \to W\ B \qquad g : W\ B \to C}{g \circ f^\dagger : W\ A \to C}$$

Example on the non-empty list (NEList) comonad.

$x \mapsto 1$		$x \mapsto \text{sum } x$		$(x \mapsto 1)^\dagger$		$(x \mapsto \text{sum } x) \circ (x \mapsto 1)^\dagger$	
$[0]$	1	$[0]$	0	$[0]$	$[1]$	$[0]$	1
$[1]$	1	$[1]$	1	$[1]$	$[1]$	$[1]$	1
$[0,0]$	1	$[0,0]$	0	$[0,0]$	$[1,1]$	$[0,0]$	2
$[0,1]$	1	$[0,1]$	1	$[0,1]$	$[1,1]$	$[0,1]$	2
$[1,0]$	1	$[1,0]$	1	$[1,0]$	$[1,1]$	$[1,0]$	2
$[1,1]$	1	$[1,1]$	2	$[1,1]$	$[1,1]$	$[1,1]$	2
...	...	...	...	...	...	...	...

Effect with monad

Monads for compositional effects:

$$\frac{f : A \to M\ B \qquad g : B \to M\ C}{?? : A \to M\ C}$$

Effect with monad

Monads for compositional effects:

$$\frac{f : A \to M\ B \qquad g^\star : M\ B \to M\ C}{g^\star \circ f : A \to M\ C}$$

Effect with monad

Monads for compositional effects:

$$\frac{f : A \to M\ B \qquad g^\star : M\ B \to M\ C}{g^\star \circ f : A \to M\ C}$$

Example on the option monad $\mathbb{N} + \{\bot\}$:

- let g := `x` $\mapsto$ `if x mod 2 = 0 then x else` $\bot$;
- let f := `x` $\mapsto$ `if x mod 3 = 0 then x else` $\bot$;
- let $g^\star$:= `y` $\mapsto$ `do x <- y; g x`;
- then $g^\star \circ f$:=

```
x ↦ if (x mod 3 = 0) and (x mod 2 = 0)
        then x
        else ⊥
```

Combine dataflow and effects

Composition:

$$\frac{f : W\ A \to M\ B \qquad\qquad g : W\ B \to M\ C}{?? : W\ A \to M\ C}$$

Combine dataflow and effects

Composition:

$$\frac{\begin{array}{ll} f : W\ A \to M\ B & g : W\ B \to M\ C \\ f^\dagger : W\ A \to \color{red}{W}\ M\ B & g^\star : \color{red}{M}\ W\ B \to M\ C \end{array}}{g^\star \circ\ \color{blue}{\delta} \circ f^\dagger : W\ A \to M\ C}$$

where

- $f^\dagger$ and $g^\star$ are reasonable extensions given the comonad W and monad M;
- δ represents the distributive law of the comonad W over the monad M, i.e.,

$$\delta : W\ M \Rightarrow M\ W$$

with 4 coherence conditions.

Example for synchronous dataflow.

Uustalu and Vene defined the distributive law:

$$\delta : (\mathbb{N} + \{\bot\})^+ \to \mathbb{N}^+ + \{\bot\}$$

that intuitively acts as:

- if the head of the list is absent (i.e., $\bot$), return absent;
- otherwise, return the list where all $\bot$ have been filtered.

Example for synchronous dataflow.

Uustalu and Vene defined the distributive law:

$$\delta : (\mathbb{N} + \{\bot\})^+ \to \mathbb{N}^+ + \{\bot\}$$

that intuitively acts as:

- if the head of the list is absent (i.e., $\bot$), return absent;
- otherwise, return the list where all $\bot$ have been filtered.

Let

- `f := (x ↦ if (head x) mod 2 = 0 then x else ⊥);`
- `g := (x ↦ sum x);`

$f^\dagger$		$g^\star$		$g^\star \circ \delta \circ f^\dagger$	
$[0]$	$[0]$	$\bot$	$\bot$	$[0]$	0
$[1]$	$[\bot]$	$[0]$	0	$[1]$	$\bot$
$[0,0]$	$[0,0]$	$[1]$	1	$[0,0]$	0
$[0,1]$	$[0,\bot]$	$[0,0]$	0	$[0,1]$	0
$[1,0]$	$[\bot,0]$	$[0,1]$	1	$[1,0]$	$\bot$
...	...	...	...	...	...

Can we extend to stateful synchronous dataflow?

Main difference with stateless case:

- the option monad is no longer sufficient as their is no notion of states;
- use of a state monad?

Stateful dataflow: first try

First idea: replace the option monad $\_ + \{\bot\}$ with the option state monad $S \to (\_ \times S) + \{\bot\}$.

But, problem for the distributive law: couldn't find one that satisfies all 4 coherence conditions.

Deadend.. After several attempts, we thought of an alternative.

Stateful dataflow: second try

Let $\mathbb{C}$ be a category, W a comonad, and M a monad in $\mathbb{C}$.

Definition (Kleisli Category)

A distributive law of W over M induces a Kleisli category $\mathbb{C}_{W,M}$, where:

- objects are objects in $\mathbb{C}$, and
- morphisms in $\mathbb{C}_{W,M}(X, Y)$ are those in $\mathbb{C}(W\ X, M\ Y)$, i.e., of the form $W\ X \to M\ Y$.

Composition and identity follow from the Kleisli category.

Stateful dataflow: second try

Let $\mathbb{C}$ be a category, W a comonad, and M a monad in $\mathbb{C}$.
Let S be a set of states.

Definition (Stateful Kleisli Category)

A distributive law of W over M induces a Stateful Kleisli category $\mathbb{C}^S_{W,M}$, where:

- objects are objects in $\mathbb{C}$, and
- morphisms in $\mathbb{C}^S_{W,M}(X, Y)$ are those in $\mathbb{C}(W\ (X \times S), M\ (Y \times S))$, i.e., of the form $W\ (X \times S) \to M\ (Y \times S)$.

Composition and identity follow from the Kleisli category.

Stateful dataflow: second try

Following Uustalu et Vene, we kept the nonempty list comonad $W = \_^+$ and the option monad $M = \_ + \{\bot\}$.

Stateful dataflow: second try

Following Uustalu et Vene, we kept the nonempty list comonad $W = \_^+$ and the option monad $M = \_ + \{\bot\}$.

Morphisms are of the form $(A \times S)^+ \to (B \times S) + \{\bot\}$.

Stateful dataflow: second try

Following Uustalu et Vene, we kept the nonempty list comonad $W = \_^+$ and the option monad $M = \_ + \{\bot\}$.

Morphisms are of the form $(A \times S)^+ \to (B \times S) + \{\bot\}$.

Lift morphisms to dataflow functions, i.e., operations on streams.

Stateful dataflow: second try

Let $f : (A \times S)^+ \to (B \times S) + \{\bot\}$ be a morphism in the stateful Kleisli category.

The run operation on f, given $\sigma \in A^\omega$ a stream of inputs in A, and s an initial state, is:

$$run(f, \sigma, s) \in (B + \{\bot\})^\omega$$

such that

- f applies to prefixes of σ with an updated state in the non-absent case;
- f applies to prefixes of σ with the same state in the absent case.

Stateful dataflow: second try

Let $f : (A \times S)^+ \to (B \times S) + \{\bot\}$ be a morphism in the stateful Kleisli category.

The run operation on f, given $\sigma \in A^\omega$ a stream of inputs in A, and s an initial state, is:

$$run(f, \sigma, s) \in (B + \{\bot\})^\omega$$

such that

- f applies to prefixes of σ with an updated state in the non-absent case;
- f applies to prefixes of σ with the same state in the absent case.

But, problem if absent: some intermediate change of states have been lost!!

Stateful dataflow: third try

We changed the option monad $M = \_ + \{\bot\}$ to the sum monad $M = \_ + E$.

Stateful dataflow: third try

We changed the option monad $M = \_ + \{\bot\}$ to the sum monad $M = \_ + E$.

We prove a new distributive law:

$$\delta : (A \times S + E)^+ \to (A \times S)^+ + E$$

where

- if the head is absent, return an exception;
- otherwise return the list where all the exceptions have been filtered.

Stateful dataflow: third try

We changed the option monad $M = \_ + \{\bot\}$ to the sum monad $M = \_ + E$.

We prove a new distributive law:

$$\delta : (A \times S + E)^+ \to (A \times S)^+ + E$$

where

- if the head is absent, return an exception;
- otherwise return the list where all the exceptions have been filtered.

The four coherence conditions have been proved in Coq.

Stateful dataflow: third try

We fix the monad to $M = \_ + S$.
Let $f : (A \times S)^+ \to (B \times S) + S$ be a morphism in stateful Kleisli category.

Run operation on f with $\sigma \in A^\omega$ a stream of inputs in A, and s an initial state:

$$run(f, \sigma, s) \in (B + \{\bot\})^\omega$$

such that

- f applies to prefixes of σ with an updated state in the non-absent case and in the absent case.

The state is propagated in case of absence!

Examples on some dataflow primitives

With a fixed S, let $A \longrightarrow B := (A \times S)^+ \to (B \times S + S)$.

Examples on some dataflow primitives

With a fixed S, let $A \longrightarrow B := (A \times S)^+ \to (B \times S + S)$.

Stateful weak product of two dataflow functions:

$$\langle f, g \rangle : A \longrightarrow (B_1 \times B_2)$$

with

- $f : A \longrightarrow B_1$
- $g : A \longrightarrow B_2$

Examples on some dataflow primitives

With a fixed S, let $A \longrightarrow B := (A \times S)^+ \to (B \times S + S)$.

Stateful weak product of two dataflow functions:

$$\langle f, g \rangle : A \longrightarrow (B_1 \times B_2)$$

with

- $f : A \longrightarrow B_1$
- $g : A \longrightarrow B_2$

Currying of dataflow function:

$$scurry\ m\ f : A \longrightarrow (B \longrightarrow C)$$

with

- $m : A^+ \times B^+ \to (A \times B)^+$
- $f : (A \times B) \longrightarrow C$

Stateful dataflow language (Syntax)

Expressions:

$$
\begin{aligned}
e ::= {} & \mathsf{tt} \mid \mathsf{true} \mid \mathsf{false} \mid n \mid e_1 \stackrel{?}{=} e_2 \mid \ldots \\
& e_1 \mathbin{;} e_2 \mid x \mid \lambda x \cdot e \mid e_1\ e_2 \mid \ldots \\
& e_1\ \mathsf{fby}\ e_2 \mid \mathsf{fix}\ n\ e_0\ e \mid \mathsf{read}\ e \mid \mathsf{write}\ e_1\ e_2
\end{aligned}
$$

Stateful dataflow language (Syntax)

Expressions:

$$
\begin{aligned}
e ::= {} & \mathsf{tt} \mid \mathsf{true} \mid \mathsf{false} \mid n \mid e_1 \stackrel{?}{=} e_2 \mid \ldots \\
& e_1 \,;\, e_2 \mid x \mid \lambda x \cdot e \mid e_1\ e_2 \mid \ldots \\
& e_1\ \mathsf{fby}\ e_2 \mid \mathsf{fix}\ n\ e_0\ e \mid \mathsf{read}\ e \mid \mathsf{write}\ e_1\ e_2
\end{aligned}
$$

Example of program:

$$
\begin{aligned}
& \mathit{first\_occurrence}\ \mathit{offset} := \\
& \lambda x \cdot \\
& \quad \mathsf{write}\ \mathit{offset}\ (1 + \mathit{offset}); \\
& \quad \mathsf{if}\ \mathit{isInList}\ \mathit{offset}\ x\ \mathsf{then}\ \bot_{\mathsf{nat}} \\
& \quad \mathsf{else} \\
& \qquad \mathsf{let}\ y\ = \mathsf{read}\ (1 + \mathit{offset})\ \mathsf{in} \\
& \qquad \mathsf{write}\ y\ x; \mathsf{write}\ (1 + \mathit{offset})\ (1 + y); x
\end{aligned}
$$

Stateful dataflow language (Semantics)

Fix the state to $S = \mathbb{N} \to \mathbb{N}$.
Fix the distributive law our new distributive law of the non-empty list comonad to the sum monad.

Stateful dataflow language (Semantics)

Fix the state to $S = \mathbb{N} \to \mathbb{N}$.
Fix the distributive law our new distributive law of the non-empty list comonad to the sum monad.

Semantics for some primitives:

$$[\![e_1 \stackrel{?}{=} e_2]\!] = \uparrow^{?}_{=} \circ \langle [\![e_1]\!], [\![e_2]\!] \rangle$$

$$[\![\text{fix } n\ e_0\ e]\!] = \begin{pmatrix} [\![e_0]\!] & \text{if } n = 0 \\ \mathit{eval} \circ \langle [\![e]\!],\ [\![\text{fix } (n-1)\ e_0\ e]\!] \rangle & \text{otherwise} \end{pmatrix}$$

where

- the equality is applied item-wise on the product $\langle [\![e_1]\!], [\![e_2]\!] \rangle$;
- the *eval* applies the result of the left part of the pair onto the right part.

Back to our example

Let

- f := (expr_semantics ((first_occurrence offset)
  (1 fby (2 fby (1 fby (0 fby (1 fby 1)))))))
- σ := sconst tt
- m_0 := `(fun a =>`

```
if (a =?  0) 1 else
if (a =?  1) 2 else
0)
```

Runnable example in Coq:

`Compute prefix 6 (run f` σ m_0`) =` $[1; 2; \bot; 0; \bot; \bot]$

Ongoing work

Program's property: mix induction (program constructs), and coinduction (program semantics).
ex: bisimulation of the run of `nat_imp` *and stream of naturals.*

Semantics property: show compositionality on runs.
The run of the composition of f and g is some form of composition of the run of f and g.

Conclusion

We presented a denotational and compositional semantics for a Stateful synchronous dataflow language.

We formalized the semantics in Coq, and provide some running examples.