Velus: towards a modular reset

Lélio Brun[1,2] – PARKAS Team

Timothy Bourke[1,2] Pierre-Évariste Dagand[4,3,1] Marc Pouzet[4,2,1]

[1]Inria Paris

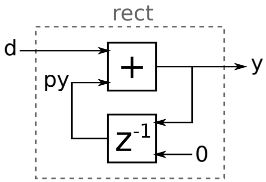
[2]DI ENS

[3]CNRS

[4]UPMC

SYNCHRON'17 — November 30, 2017

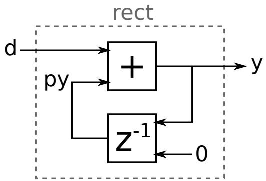
Lustre[1]: example



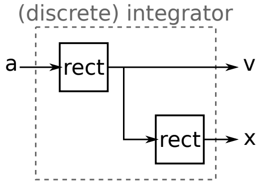

```
node rect(d: int) returns (y: int)
  var py: int;
let
  y = py + d;
  py = 0 fby y;
tel
```

[1] Caspi et al. (1987): "LUSTRE: A declarative language for programming synchronous systems"

Lustre: example

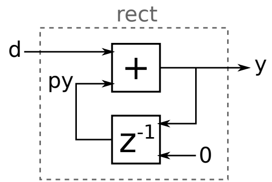



```
node rect(d: int) returns (y: int)
  var py: int;
let
  y = py + d;
  py = 0 fby y;
tel
```

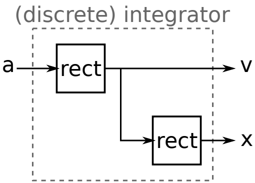



```
node integrator(a: int) returns (v, x: int)
let
  v = rect(a);
  x = rect(v);
tel
```

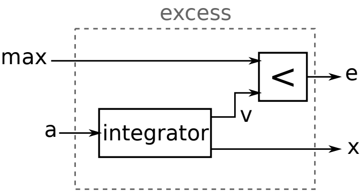
Lustre: example




```
node rect(d: int) returns (y: int)
  var py: int;
let
  y = py + d;
  py = 0 fby y;
tel
```




```
node integrator(a: int) returns (v, x: int)
let
  v = rect(a);
  x = rect(v);
tel
```




```
node excess(max, a: int)
  returns (e: bool; x: int)
  var v: int;
let
  (v, x) = integrator(a);
  e = max < v;
tel
```

Context

Critical aspect

- specification norms (DO-178B), industrial certification

Context

Critical aspect

- specification norms (DO-178B), industrial certification
- formal verification, mechanized proofs, proof assistant (eg. Coq[1])

[1] The Coq Development Team (2016): *The Coq proof assistant reference manual*

Context

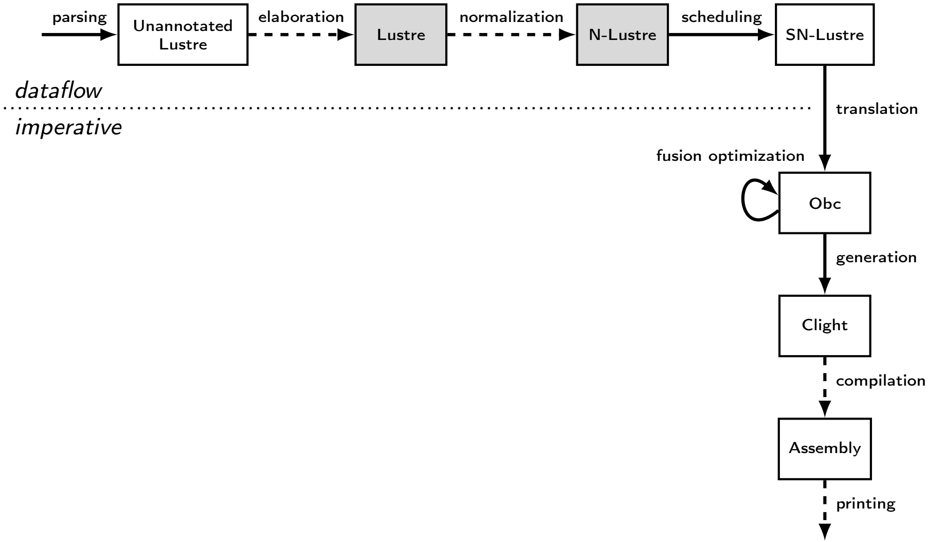
Critical aspect

- specification norms (DO-178B), industrial certification
- formal verification, mechanized proofs, proof assistant (eg. Coq)

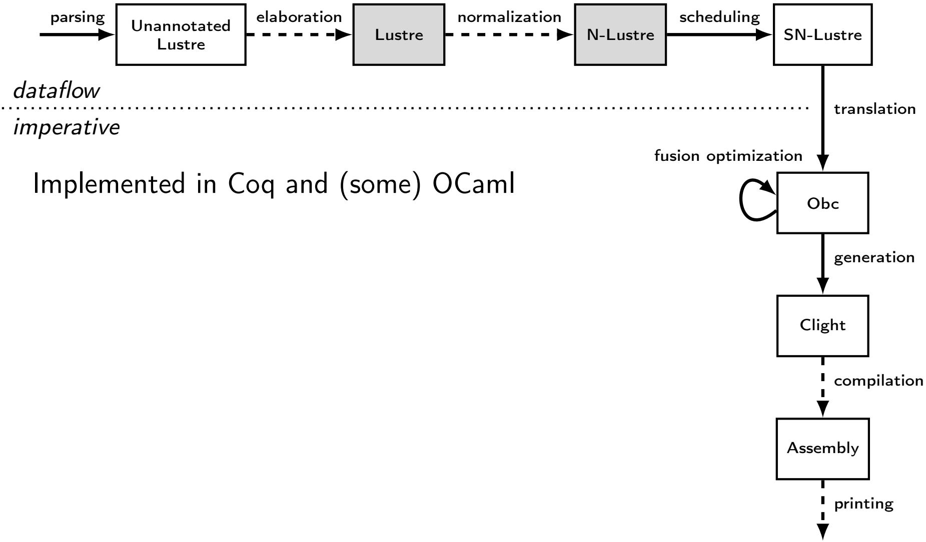
Goal

Develop a formally verified code generator

Vélus: a verified compiler

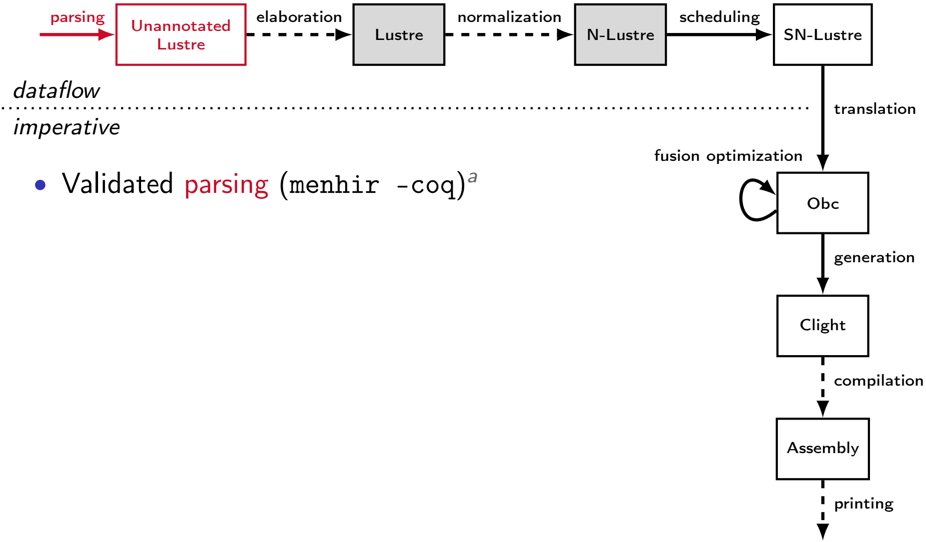


Vélus: a verified compiler



Implemented in Coq and (some) OCaml

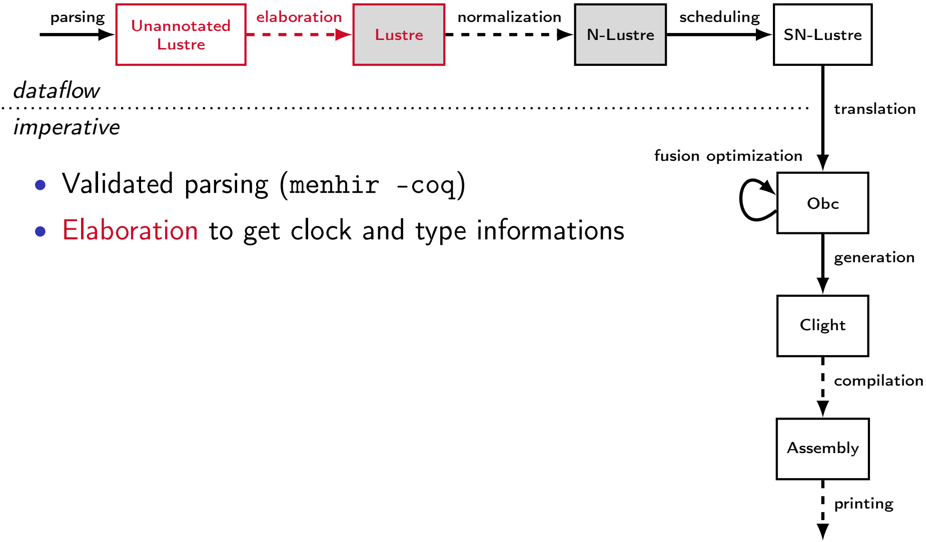
Vélus: a verified compiler



- Validated parsing (`menhir -coq`)[a]

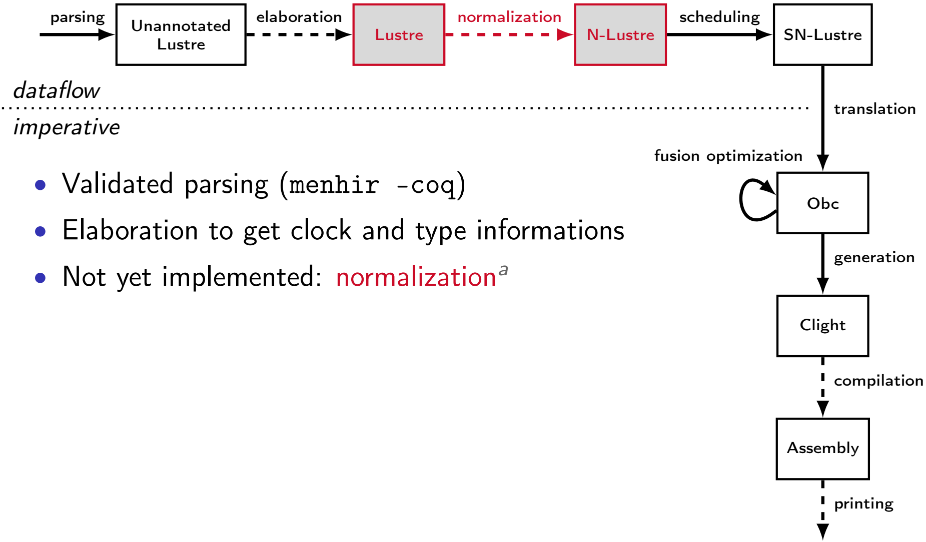
[a] Jourdan, Pottier, and Leroy (2012): "Validating LR(1) parsers"

Vélus: a verified compiler



- Validated parsing (`menhir -coq`)
- Elaboration to get clock and type informations

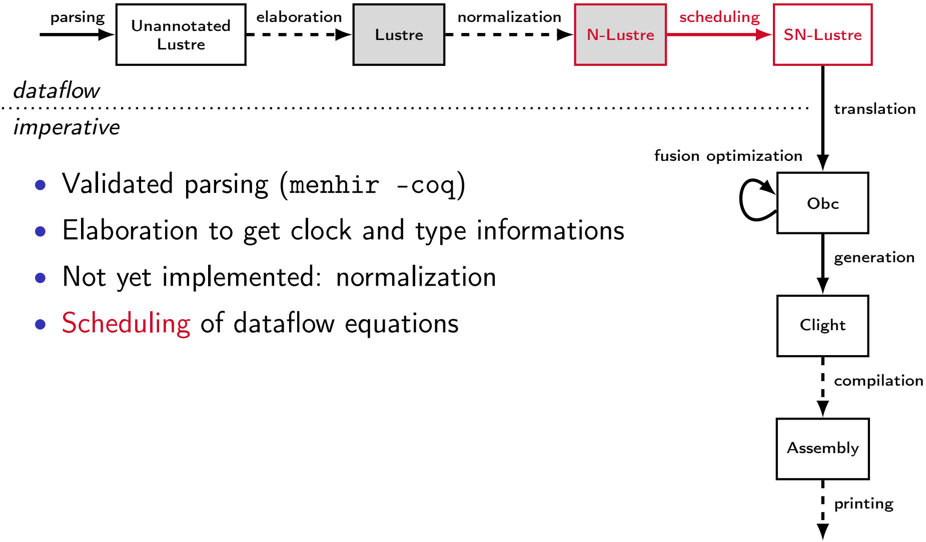
Vélus: a verified compiler



- Validated parsing (`menhir -coq`)
- Elaboration to get clock and type informations
- Not yet implemented: normalization[a]

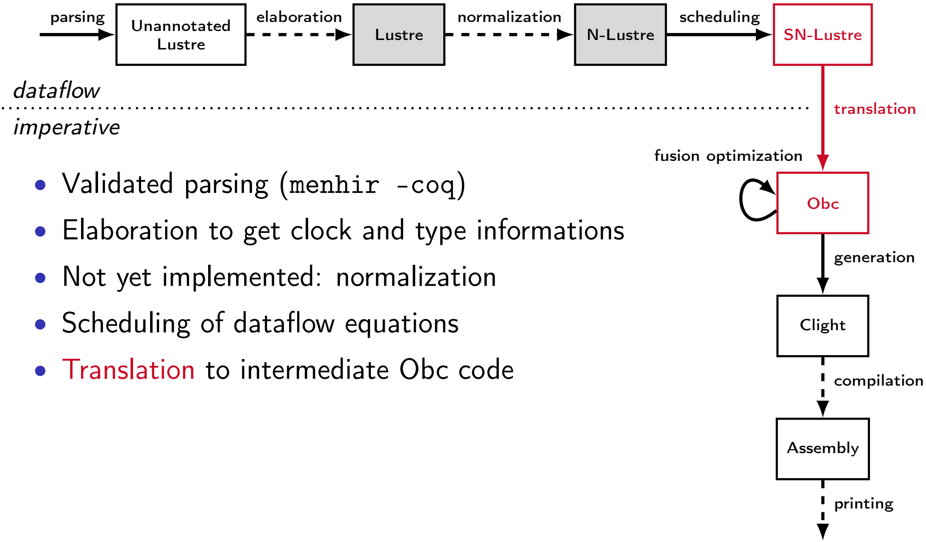
[a] Auger (2013): "Compilation certifiée de SCADE/LUSTRE"

Vélus: a verified compiler



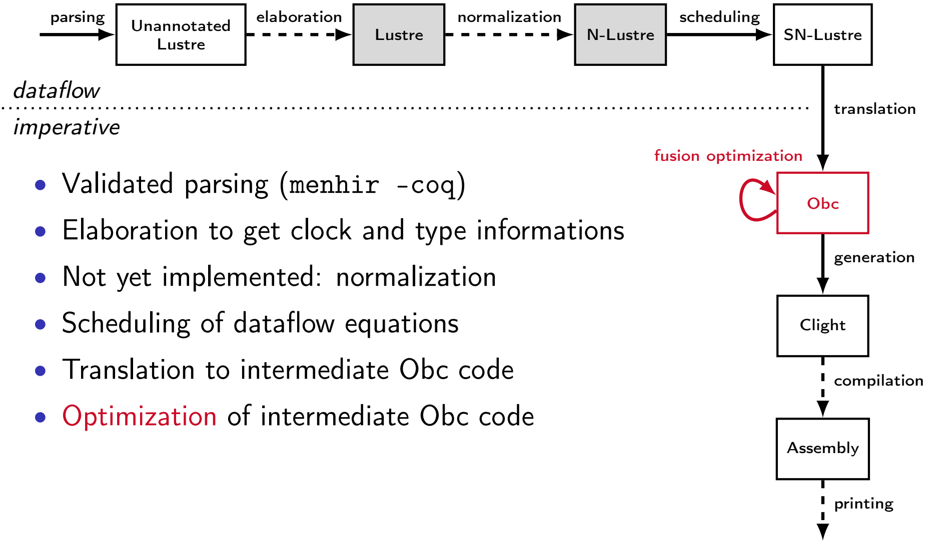
- Validated parsing (`menhir -coq`)
- Elaboration to get clock and type informations
- Not yet implemented: normalization
- Scheduling of dataflow equations

Vélus: a verified compiler



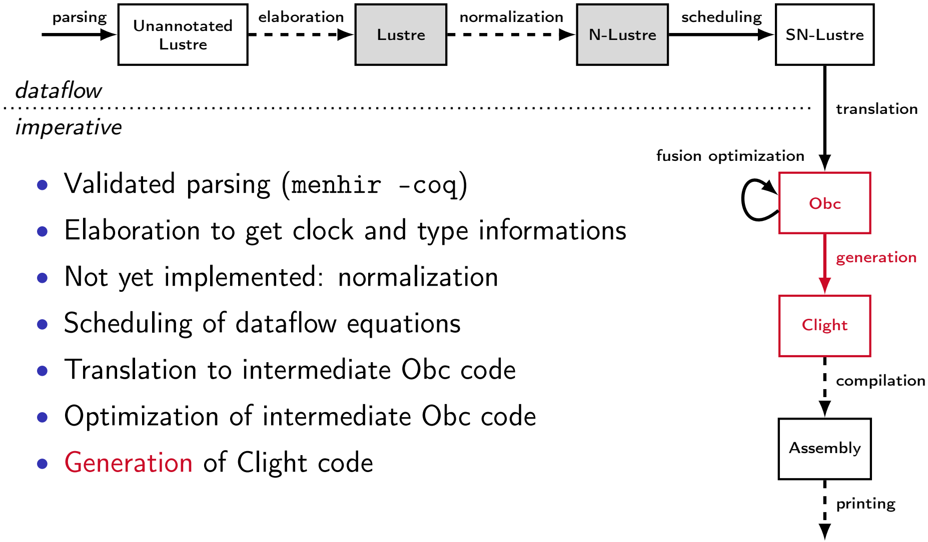
- Validated parsing (`menhir -coq`)
- Elaboration to get clock and type informations
- Not yet implemented: normalization
- Scheduling of dataflow equations
- Translation to intermediate Obc code

Vélus: a verified compiler



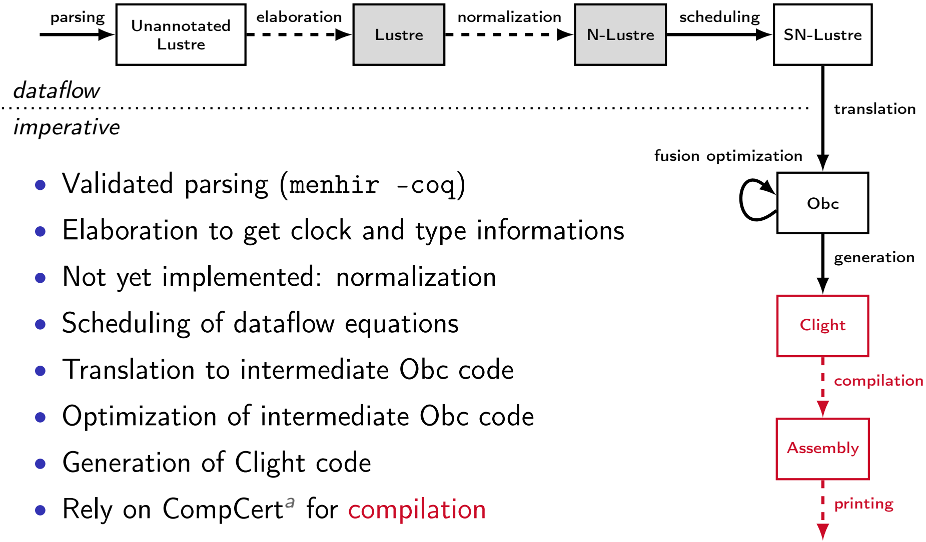
- Validated parsing (`menhir -coq`)
- Elaboration to get clock and type informations
- Not yet implemented: normalization
- Scheduling of dataflow equations
- Translation to intermediate Obc code
- Optimization of intermediate Obc code

Vélus: a verified compiler



- Validated parsing (`menhir -coq`)
- Elaboration to get clock and type informations
- Not yet implemented: normalization
- Scheduling of dataflow equations
- Translation to intermediate Obc code
- Optimization of intermediate Obc code
- Generation of Clight code

Vélus: a verified compiler



- Validated parsing (`menhir -coq`)
- Elaboration to get clock and type informations
- Not yet implemented: normalization
- Scheduling of dataflow equations
- Translation to intermediate Obc code
- Optimization of intermediate Obc code
- Generation of Clight code
- Rely on CompCert[a] for compilation

[a] Blazy, Dargaye, and Leroy (2006): "Formal verification of a C compiler front-end"

N-Lustre: abstract syntax

$le :=$	**expression**
$\mid\ k$	(constant)
$\mid\ x$	(variable)
$\mid\ le$ `when` x	(when)
$\mid\ op\ e$	(unary operator)
$\mid\ e\ bop\ e$	(binary operator)

$ce :=$	**control expression**
$\mid$ `merge` $x\ ce\ ce$	(merge)
$\mid$ `if` x `then` ce `else` ce	(if)
$\mid\ le$	(expression)

$eq :=$	**equation**
$\mid\ x :: c = ce$	(def)
$\mid\ x :: c = k$ `fby` le	(fby)
$\mid\ \overrightarrow{x} :: c = x(\overrightarrow{le})$	(app)
$\mid\ \overrightarrow{x} :: c = x(\overrightarrow{le})$ `every` x	(reset)

$n :=$ **node**

$\mid$ `node` $x(\overrightarrow{x^{ty::c}})$ `returns` $(\overrightarrow{x^{ty::c}})$

[var $\overrightarrow{x^{ty::c}}$]

`let`

$\overrightarrow{eq}_{;}$

`tel`

Lustre semantics in Coq

streams as maps $\mathbb{N} \rightarrow$ value

- not as direct as in literature
- maybe too difficult to adapt to Lustre

Lustre semantics in Coq

streams as maps $\mathbb{N} \rightarrow$ value

- not as direct as in literature
- maybe too difficult to adapt to Lustre

Goal

Propose a new formalization based on co-inductive

- suitable to our proofs
- as a reference semantics for Lustre
- suitable to do verification of programs

Lustre semantics in Coq

streams as maps $\mathbb{N} \rightarrow$ value

- not as direct as in literature
- maybe too difficult to adapt to Lustre

Goal

Propose a new formalization based on co-inductive

- suitable to our proofs
- as a reference semantics for Lustre
- suitable to do verification of programs

Bonus

Formalize the semantics of the modular reset

N-Lustre semantics

k `fby` x

$$\mathrm{fby}^{\#}\ k\ \ (\bot \cdot xs) = \bot \cdot \mathrm{fby}^{\#}\ k\ xs$$
$$\mathrm{fby}^{\#}\ k\ \ (x \cdot xs) = k \cdot \mathrm{fby}^{\#}\ x\ xs$$

N-Lustre semantics

k `fby` x

$$\text{fby}^{\#}\ k\ \ (\bot \cdot xs) = \bot \cdot \text{fby}^{\#}\ k\ xs$$
$$\text{fby}^{\#}\ k\ \ (x \cdot xs) = k \cdot \text{fby}^{\#}\ x\ xs$$

x `when` c

$$\text{when}^{\#}\ \ (\bot \cdot xs)\ \ (\bot \cdot cs) = \bot \cdot \text{when}^{\#}\ xs\ cs$$
$$\text{when}^{\#}\ \ (x \cdot xs)\ \ (\text{false} \cdot cs) = \bot \cdot \text{when}^{\#}\ xs\ cs$$
$$\text{when}^{\#}\ \ (x \cdot xs)\ \ (\text{true} \cdot cs) = x \cdot \text{when}^{\#}\ xs\ cs$$

N-Lustre semantics

`if` x `then` t `else` f

$$\mathrm{ite}^{\#}\ (\bot \cdot xs)\ (\bot \cdot ts)\ (\bot \cdot fs) = \bot \cdot \mathrm{ite}^{\#}\ xs\ ts\ fs$$
$$\mathrm{ite}^{\#}\ (\mathrm{true} \cdot xs)\ (x \cdot ts)\ (y \cdot fs) = x \cdot \mathrm{ite}^{\#}\ xs\ ts\ fs$$
$$\mathrm{ite}^{\#}\ (\mathrm{false} \cdot xs)\ (x \cdot ts)\ (y \cdot fs) = y \cdot \mathrm{ite}^{\#}\ xs\ ts\ fs$$

N-Lustre semantics

`if` x `then` t `else` f

$$\begin{aligned}
\text{ite}^{\#}\ (\bot \cdot xs)\ (\bot \cdot ts)\ (\bot \cdot fs) &= \bot \cdot \text{ite}^{\#}\ xs\ ts\ fs \\
\text{ite}^{\#}\ (\text{true} \cdot xs)\ (x \cdot ts)\ (y \cdot fs) &= x \cdot \text{ite}^{\#}\ xs\ ts\ fs \\
\text{ite}^{\#}\ (\text{false} \cdot xs)\ (x \cdot ts)\ (y \cdot fs) &= y \cdot \text{ite}^{\#}\ xs\ ts\ fs
\end{aligned}$$

`merge` $c\ x\ y$

$$\begin{aligned}
\text{merge}^{\#}\ (\bot \cdot cs)\ (\bot \cdot xs)\ (\bot \cdot ys) &= \bot \cdot \text{merge}^{\#}\ cs\ xs\ ys \\
\text{merge}^{\#}\ (\text{true} \cdot cs)\ (x \cdot xs)\ (\bot \cdot ys) &= x \cdot \text{merge}^{\#}\ cs\ xs\ ys \\
\text{merge}^{\#}\ (\text{false} \cdot cs)\ (\bot \cdot xs)\ (y \cdot ys) &= y \cdot \text{merge}^{\#}\ cs\ xs\ ys
\end{aligned}$$

Streams formalization in Coq

- maps $\mathbb{N} \rightarrow$ value
  - not as direct as in literature
  - maybe too difficult to adapt to Lustre
  - rather intricate to formalize

Streams formalization in Coq

- maps $\mathbb{N} \rightarrow$ value
  - not as direct as in literature
  - maybe too difficult to adapt to Lustre
  - rather intricate to formalize
- co-inductive streams

```
CoInductive Stream {A : Type} : Type :=
  Cons : A → Stream → Stream.
Infix "·" := Cons.
```

Streams formalization in Coq

- maps $\mathbb{N} \to$ value
  - not as direct as in literature
  - maybe too difficult to adapt to Lustre
  - rather intricate to formalize
- co-inductive streams

```
CoInductive Stream {A : Type} : Type :=
  Cons : A → Stream → Stream.
Infix "·" := Cons.
```

```
Fixpoint hd {A : Type} (s : Stream A) : A :=
  match s with
    x · _ ⇒ x
  end.
```

Streams formalization in Coq

- maps $\mathbb{N} \rightarrow$ value
  - not as direct as in literature
  - maybe too difficult to adapt to Lustre
  - rather intricate to formalize
- co-inductive streams

```
CoInductive Stream {A : Type} : Type :=
  Cons : A → Stream → Stream.
Infix "·" := Cons.

Fixpoint hd {A : Type} (s : Stream A) : A :=
  match s with
    x · _ ⇒ x
  end.

CoFixpoint map {A B : Type} (f : A → B) (s : Stream A) :
    Stream B :=
  match s with
    x · s ⇒ f x · map f s
  end.
```

Co-inductive streams based semantics

- classical Lustre formalization for values

```
Inductive value :=
| absent
| present (c : val).

Definition vstream := Stream value.
```

Co-inductive streams based semantics

- classical Lustre formalization for values

```
Inductive value :=
| absent
| present (c : val).

Definition vstream := Stream value.
```

- parameters: environment and base clock

$$\frac{H, b \vdash_{\mathsf{le}} k \Rightarrow ks}{H, \mathsf{T} \cdot b \vdash_{\mathsf{le}} k \Rightarrow \mathsf{sem\_const}\ k \cdot ks} \qquad \frac{H, b \vdash_{\mathsf{le}} k \Rightarrow ks}{H, \mathsf{F} \cdot b \vdash_{\mathsf{le}} k \Rightarrow \bot \cdot ks}$$

```
Definition history := PM.t vstream.
Definition clock := Stream bool.

CoFixpoint const (k: const) (b: clock): vstream :=
  match b with
  | true  · b' ⇒ present (sem_const k) · const k b'
  | false · b' ⇒ absent                · const k b'
  end.
```

Co-inductive streams based semantics: dealing with equality

$$\frac{H(x) \equiv xs}{H, b \vdash_{\mathsf{le}} x \Rightarrow xs}$$

```
CoInductive EqSt {A : Type} (s1 s2: Stream A) : Prop :=
  eqst:
    hd s1 = hd s2 →
    EqSt (tl s1) (tl s2) →
    EqSt s1 s2.
Infix "≡" := EqSt.

Inductive sem_var: history → ident → vstream → Prop :=
  sem_var_intro:
    ∀ H x xs xs',
      PM.MapsTo x xs' H →
      xs ≡ xs' →
      sem_var H x xs.
```

Co-inductive streams based semantics: k `fby` x

$$\mathsf{fby}^{\#}\ k\ \ (\bot \cdot xs) = \bot \cdot \mathsf{fby}^{\#}\ k\ xs$$

$$\mathsf{fby}^{\#}\ k\ \ (x \cdot xs) = k \cdot \mathsf{fby}^{\#}\ x\ xs$$

```
CoFixpoint fby (k: val) (xs: vstream) : vstream :=
  match xs with
  | absent    · xs ⇒ absent    · fby k xs
  | present x · xs ⇒ present k · fby x xs
  end.
```

Co-inductive streams based semantics: x `when` c

$$\text{when}^{\#}\ (\bot \cdot xs)\ (\bot \cdot cs) = \bot \cdot \text{when}^{\#}\ xs\ cs$$
$$\text{when}^{\#}\ (x \cdot xs)\ (\text{false} \cdot cs) = \bot \cdot \text{when}^{\#}\ xs\ cs$$
$$\text{when}^{\#}\ (x \cdot xs)\ (\text{true} \cdot cs) = x \cdot \text{when}^{\#}\ xs\ cs$$

```
CoInductive when : vstream → vstream → vstream → Prop :=
| WhenA:
    ∀ xs cs rs,
      when xs cs rs →
      when (absent · xs) (absent · cs) (absent · rs)
| WhenPA:
    ∀ x c xs cs rs,
      when xs cs rs →
      when (present x · xs) (present false_val · cs) (absent · rs)
| WhenPP:
    ∀ x c xs cs rs,
      when xs cs rs →
      when (present x · xs) (present true_val · cs) (present x · rs
    ).
```

Co-inductive streams based semantics: `if` c `then` x `else` y

$$\mathsf{ite}^{\#}\ (\bot \cdot xs)\ (\bot \cdot ts)\ (\bot \cdot fs) = \bot \cdot \mathsf{ite}^{\#}\ xs\ ts\ fs$$
$$\mathsf{ite}^{\#}\ (\mathsf{true} \cdot xs)\ (x \cdot ts)\ (y \cdot fs) = x \cdot \mathsf{ite}^{\#}\ xs\ ts\ fs$$
$$\mathsf{ite}^{\#}\ (\mathsf{false} \cdot xs)\ (x \cdot ts)\ (y \cdot fs) = y \cdot \mathsf{ite}^{\#}\ xs\ ts\ fs$$

```
CoInductive ite : vstream → vstream → vstream → vstream → Prop :=
| IteA:
    ∀ s ts fs rs,
      ite s ts fs rs →
      ite (absent · s) (absent · ts) (absent · fs) (absent · rs)
| IteT:
    ∀ t f s ts fs rs,
      ite s ts fs rs →
      ite (present true_val · s)
          (present t · ts) (present f · fs) (present t · rs)
| IteF:
    ∀ t f s ts fs rs,
      ite s ts fs rs →
      ite (present false_val · s)
          (present t · ts) (present f · fs) (present f · rs).
```

Co-inductive streams based semantics: merge $c\ x\ y$

$$\text{merge}^{\#}\ (\bot \cdot cs)\ (\bot \cdot xs)\ (\bot \cdot ys) = \bot \cdot \text{merge}^{\#}\ cs\ xs\ ys$$
$$\text{merge}^{\#}\ (\text{true} \cdot cs)\ (x \cdot xs)\ (\bot \cdot ys) = x \cdot \text{merge}^{\#}\ cs\ xs\ ys$$
$$\text{merge}^{\#}\ (\text{false} \cdot cs)\ (\bot \cdot xs)\ (y \cdot ys) = y \cdot \text{merge}^{\#}\ cs\ xs\ ys$$

```
CoInductive merge : vstream → vstream → vstream → vstream → Prop :=
| MergeA:
    ∀ cs xs ys rs,
      merge cs xs ys rs →
      merge (absent · cs) (absent · xs) (absent · ys) (absent · rs)
| MergeT:
    ∀ t cs xs ys rs,
      merge cs xs ys rs →
      merge (present true_val · cs)
            (present t · xs) (absent · ys) (present t · rs)
| MergeF:
    ∀ f cs xs ys rs,
      merge cs xs ys rs →
      merge (present false_val · cs)
            (absent · xs) (present f · ys) (present f · rs).
```

Co-inductive streams based inductive semantics

Expressions

```
Inductive sem_lexp: history → clock → lexp → vstream → Prop :=
| Sconst:
    ∀ H b c cs,
      cs ≡ const c b →
      sem_lexp H b (Econst c) cs
| Svar:
    ∀ H b x ty xs,
      sem_var H x xs →
      sem_lexp H b (Evar x ty) xs
| Swhen:
    ∀ H b e x k es xs os,
      sem_lexp H b e es →
      sem_var H x xs →
      when k es xs os →
      sem_lexp H b (Ewhen e x k) os
| Sunop:
    ∀ H b op e ty es os,
      sem_lexp H b e es →
      lift1 op (typeof e) es os →
      sem_lexp H b (Eunop op e ty) os
| Sbinop:
    ∀ H b op e1 e2 ty es1 es2 os,
      sem_lexp H b e1 es1 →
      sem_lexp H b e2 es2 →
      lift2 op (typeof e1) (typeof e2) es1 es2 os →
      sem_lexp H b (Ebinop op e1 e2 ty) os.
```

Co-inductive streams based inductive semantics

Control expressions

```
Inductive sem_cexp: history → clock → cexp → vstream → Prop :=
| Smerge:
    ∀ H b x t f xs ts fs os,
      sem_var H x xs →
      sem_cexp H b t ts →
      sem_cexp H b f fs →
      merge xs ts fs os →
      sem_cexp H b (Emerge x t f) os
| Site:
    ∀ H b e t f es ts fs os,
      sem_lexp H b e es →
      sem_cexp H b t ts →
      sem_cexp H b f fs →
      ite es ts fs os →
      sem_cexp H b (Eite e t f) os
| Sexp:
    ∀ H b e es,
      sem_lexp H b e es →
      sem_cexp H b (Eexp e) es.
```

Co-inductive streams based inductive semantics

Equations and nodes

```
Inductive sem_equation: history → clock → equation → Prop :=
| SeqDef:
    ∀ H b x ck e es,
      sem_cexp H b e es →
      sem_var H x es →
      sem_equation H b (EqDef x ck e)
| SeqFby:
    ∀ H b x ck c0 e es os,
      sem_lexp H b e es →
      os = fby (sem_const c0) es →
      sem_var H x os →
      sem_equation H b (EqFby x ck c0 e)
| SeqApp:
    ∀ H b ys ck f es ess oss,
      Forall2 (sem_lexp H b) es ess →
      sem_node f ess oss →
      Forall2 (sem_var H) ys oss →
      sem_equation H b (EqApp ys ck f es)

with sem_node: ident → list vstream → list vstream → Prop :=
     SNode:
       ∀ H f n xss oss,
         find_node f G = Some n →
         Forall2 (sem_var H) (idents n.(n_in)) xss →
         Forall2 (sem_var H) (idents n.(n_out)) oss →
         Forall (sem_equation H (clocks_of xss)) n.(n_eqs) →
         sem_node f xss oss.
```

Adding the modular reset

- node application: $f(x_0, \ldots, x_n)$
  call the node f

Adding the modular reset

- node application: $f(x_0, \ldots, x_n)$
  call the node f
- modular reset: $f(x_0, \ldots, x_n)$ every c
  reset the internal state (delays) of f at each tick of c

Adding the modular reset

- node application: $f(x_0, \ldots, x_n)$
  call the node f
- modular reset: $f(x_0, \ldots, x_n)$ every c
  reset the internal state (delays) of f at each tick of c

r	F
x	x_0
$f(x)$	y_0
$f(x)$ every r	y_0

Adding the modular reset

- node application: $f(x_0, \ldots, x_n)$
  call the node f
- modular reset: $f(x_0, \ldots, x_n)$ every c
  reset the internal state (delays) of f at each tick of c

r	F	F
x	x_0	x_1
$f(x)$	y_0	y_1
$f(x)$ every r	y_0	y_1

Adding the modular reset

- node application: $f(x_0, \ldots, x_n)$
  call the node f
- modular reset: $f(x_0, \ldots, x_n)$ every c
  reset the internal state (delays) of f at each tick of c

r	F	F	T
x	x_0	x_1	x_2
$f(x)$	y_0	y_1	y_2
$f(x)$ every r	y_0	y_1	y_2'

Adding the modular reset

- node application: $f(x_0, \ldots, x_n)$
  call the node f
- modular reset: $f(x_0, \ldots, x_n)$ every c
  reset the internal state (delays) of f at each tick of c

r	F	F	T	F
x	x_0	x_1	x_2	x_3
$f(x)$	y_0	y_1	y_2	y_3
$f(x)$ every r	y_0	y_1	y_2'	y_3'

Adding the modular reset

- node application: $f(x_0, \dots, x_n)$
  call the node f
- modular reset: $f(x_0, \dots, x_n)$ every c
  reset the internal state (delays) of f at each tick of c

r	F	F	T	F	F
x	x_0	x_1	x_2	x_3	x_4
$f(x)$	y_0	y_1	y_2	y_3	y_4
$f(x)$ every r	y_0	y_1	y'_2	y'_3	y'_4

Adding the modular reset

- node application: $f(x_0, \ldots, x_n)$
  call the node f
- modular reset: $f(x_0, \ldots, x_n)$ every c
  reset the internal state (delays) of f at each tick of c

r	F	F	T	F	F	T
x	x_0	x_1	x_2	x_3	x_4	x_5
$f(x)$	y_0	y_1	y_2	y_3	y_4	y_5
$f(x)$ every r	y_0	y_1	y_2'	y_3'	y_4'	y_5''

Adding the modular reset

- node application: $f(x_0, \ldots, x_n)$
  call the node f
- modular reset: $f(x_0, \ldots, x_n)$ every c
  reset the internal state (delays) of f at each tick of c

r	F	F	T	F	F	T	F
x	x_0	x_1	x_2	x_3	x_4	x_5	x_6
$f(x)$	y_0	y_1	y_2	y_3	y_4	y_5	y_6
$f(x)$ every r	y_0	y_1	y'_2	y'_3	y'_4	y''_5	y''_6

Adding the modular reset

- node application: $f(x_0, \ldots, x_n)$
  call the node f
- modular reset: $f(x_0, \ldots, x_n)$ every c
  reset the internal state (delays) of f at each tick of c

r	F	F	T	F	F	T	F	T
x	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7
$f(x)$	y_0	y_1	y_2	y_3	y_4	y_5	y_6	y_7
$f(x)$ every r	y_0	y_1	y_2'	y_3'	y_4'	y_5''	y_6''	y_7'''

Adding the modular reset

- node application: $f(x_0, \ldots, x_n)$
  call the node f
- modular reset: $f(x_0, \ldots, x_n)$ every c
  reset the internal state (delays) of f at each tick of c

r	F	F	T	F	F	T	F	T	F
x	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8
$f(x)$	y_0	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8
$f(x)$ every r	y_0	y_1	y_2'	y_3'	y_4'	y_5''	y_6''	y_7'''	y_8'''

Adding the modular reset

- node application: $f(x_0, \ldots, x_n)$
  call the node f
- modular reset: $f(x_0, \ldots, x_n)$ every c
  reset the internal state (delays) of f at each tick of c

r	F	F	T	F	F	T	F	T	F	...
x	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	...
$f(x)$	y_0	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8	...
$f(x)$ every r	y_0	y_1	y_2'	y_3'	y_4'	y_5''	y_6''	y_7'''	y_8'''	...

Semantics?

A recursive intuition, not valid definition in Lustre[1]

```
node true_until(c: bool) returns (x: bool)
let
  x = true fby (if c then false else x);
tel

node reset_f(x: int, r: bool) returns (y: int)
  vars c: bool;
let
  c = true_until(r);
  y = merge c (f(x when c))
              (reset_f(x when not c, r when not c));
tel
```

[1] Hamon and Pouzet (2000): "Modular Resetting of Synchronous Data-flow Programs"

Semantics?

A recursive intuition, not valid definition in Lustre[1]

```
node true_until(c: bool) returns (x: bool)
let
  x = true fby (if c then false else x);
tel

node reset_f(x: int, r: bool) returns (y: int)
  vars c: bool;
let
  c = true_until(r);
  y = merge c (f(x when c))
              (reset_f(x when not c, r when not c));
tel
```

Writable in Coq but as an intricate co-inductive predicate: we need another solution

[1]Hamon and Pouzet (2000): "Modular Resetting of Synchronous Data-flow Programs"

Infinitely unrolling the recursion

r	F	F	T	F	F	T	F	T	F	$\dots$
x	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	$\dots$

Infinitely unrolling the recursion

mask: a cofixpoint written in Coq

r	F	F	T	F	F	T	F	T	F	...
x	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	...
mask 0 r x	x_0	x_1	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	...
f(mask 0 r x)	y_0	y_1	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	...

Infinitely unrolling the recursion

mask: a cofixpoint written in Coq

r	F	F	T	F	F	T	F	T	F	...
x	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	...
mask 0 r x	x_0	x_1	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	...
f(mask 0 r x)	y_0	y_1	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	...
mask 1 r x	$\bot$	$\bot$	x_2	x_3	x_4	$\bot$	$\bot$	$\bot$	$\bot$	...
f(mask 1 r x)	$\bot$	$\bot$	y'_2	y'_3	y'_4	$\bot$	$\bot$	$\bot$	$\bot$	...

Infinitely unrolling the recursion

mask: a cofixpoint written in Coq

r	F	F	T	F	F	T	F	T	F	...
x	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	...
mask 0 r x	x_0	x_1	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	...
f (mask 0 r x)	y_0	y_1	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	...
mask 1 r x	$\perp$	$\perp$	x_2	x_3	x_4	$\perp$	$\perp$	$\perp$	$\perp$	...
f (mask 1 r x)	$\perp$	$\perp$	y'_2	y'_3	y'_4	$\perp$	$\perp$	$\perp$	$\perp$	...
mask 2 r x	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	x_5	x_6	$\perp$	$\perp$	...
f (mask 2 r x)	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	y''_5	y''_6	$\perp$	$\perp$	...

Infinitely unrolling the recursion

mask: a cofixpoint written in Coq

r	F	F	T	F	F	T	F	T	F	...
x	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	...
mask 0 r x	x_0	x_1	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	...
f(mask 0 r x)	y_0	y_1	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	...
mask 1 r x	$\bot$	$\bot$	x_2	x_3	x_4	$\bot$	$\bot$	$\bot$	$\bot$	...
f(mask 1 r x)	$\bot$	$\bot$	y_2'	y_3'	y_4'	$\bot$	$\bot$	$\bot$	$\bot$	...
mask 2 r x	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	x_5	x_6	$\bot$	$\bot$	...
f(mask 2 r x)	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	y_5''	y_6''	$\bot$	$\bot$	...
mask 3 r x	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	x_7	x_8	...
f(mask 3 r x)	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	y_7'''	y_8'''	...

Infinitely unrolling the recursion

mask: a cofixpoint written in Coq

r	F	F	T	F	F	T	F	T	F	...
x	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	...
mask 0 r x	x_0	x_1	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	...
f(mask 0 r x)	y_0	y_1	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	...
mask 1 r x	$\bot$	$\bot$	x_2	x_3	x_4	$\bot$	$\bot$	$\bot$	$\bot$	...
f(mask 1 r x)	$\bot$	$\bot$	y'_2	y'_3	y'_4	$\bot$	$\bot$	$\bot$	$\bot$	...
mask 2 r x	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	x_5	x_6	$\bot$	$\bot$	...
f(mask 2 r x)	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	y''_5	y''_6	$\bot$	$\bot$	...
mask 3 r x	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	x_7	x_8	...
f(mask 3 r x)	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	y'''_7	y'''_8	...
$\vdots$										

Infinitely unrolling the recursion

mask: a cofixpoint written in Coq

r	F	F	T	F	F	T	F	T	F	...
x	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	...
mask 0 r x	x_0	x_1	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	...
f(mask 0 r x)	y_0	y_1	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	...
mask 1 r x	$\bot$	$\bot$	x_2	x_3	x_4	$\bot$	$\bot$	$\bot$	$\bot$	...
f(mask 1 r x)	$\bot$	$\bot$	y'_2	y'_3	y'_4	$\bot$	$\bot$	$\bot$	$\bot$	...
mask 2 r x	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	x_5	x_6	$\bot$	$\bot$	...
f(mask 2 r x)	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	y''_5	y''_6	$\bot$	$\bot$	...
mask 3 r x	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	x_7	x_8	...
f(mask 3 r x)	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	y'''_7	y'''_8	...
$\vdots$										
$f(x)$ every r	y_0	y_1	y'_2	y'_3	y'_4	y''_5	y''_6	y'''_7	y'''_8	...

Formal semantics

Use of an universally quantified relation as a constraint

```
Inductive sem_equation : history → clock → equation → Prop :=
...
| SeqApp:
    ∀ H b ys ck f es ess oss,
      ...
      sem_equation H b (EqApp ys ck f es None)
| SeqReset:
    ∀ H b ys ck f es x xs ess oss,
      Forall2 (sem_lexp H b) es ess →
      sem_var H x xs →
      sem_reset f (reset_of xs) ess oss →
      Forall2 (sem_var H) ys oss →
      sem_equation H b (EqApp ys ck f es (Some x))
...

with sem_node : ident → list vstream → list vstream → Prop := ...

with sem_reset : ident → clock → list vstream → list vstream → Prop :=
        SReset:
          ∀ f r xss yss,
            (∀ n, sem_node f (List.map (mask n r) xss) (List.map (mask n r) yss)) →
            sem_reset f r xss yss.
```

Summary

- co-inductive based semantics
  - more direct, more natural
  - notoriously hard co-inductive proofs, unfamous `cofix`
  - extensible to un-normalized Lustre
- very neat semantics for modular reset

Future work

- compilation
- automata
- mix best of both worlds?

Other work

- synchronous languages, Lustre [Cas+87; Aug13; Ben+03; Bie+08; Aug+14; Bou+16]
- verified compilation: CompCert [BDL06; Ler09a; Ler09b]
- automatic proof of a compiler [CG15]
- denotational semantics [Chl07; BKV09; BH09]

References I

[Cas+87] Paul Caspi, Nicolas Halbwachs, Daniel Pilaud, and John Alexander Plaice. "LUSTRE: A declarative language for programming synchronous systems." In: *POPL'87*. ACM. Jan. 1987, pp. 178–188.

[The16] The Coq Development Team. *The Coq proof assistant reference manual*. Version 8.5. Inria. 2016. url: `http://coq.inria.fr`.

[JPL12] Jacques-Henri Jourdan, François Pottier, and Xavier Leroy. "Validating LR(1) parsers." In: *21st European Symposium on Programming (ESOP 2012), held as part of European Joint Conferences on Theory and Practice of Software (ETAPS 2012)*. Ed. by Helmut Seidl. Vol. 7211. Lecture Notes in Comp. Sci. Tallinn, Estonia: Springer, Mar. 2012, pp. 397–416.

[Aug13] Cédric Auger. "Compilation certifiée de SCADE/LUSTRE." PhD thesis. Orsay, France: Univ. Paris Sud 11, Apr. 2013.

[BDL06] Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. "Formal verification of a C compiler front-end." In: *FM 2006: Int. Symp. on Formal Methods* volume 4085 de LNCS (2006), pp. 460–475.

References II

[HP00] Grégoire Hamon and Marc Pouzet. "Modular Resetting of Synchronous Data-flow Programs." In: *ACM International conference on Principles of Declarative Programming (PPDP'00)*. Montreal, Canada, Sept. 2000. url: `ppdp00.ps.gz`.

[Ben+03] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Paul Le Guernic, Nicolas Halbwachs, and Robert De Simone. "The synchronous languages 12 years later." In: *proceedings of the IEEE*. Vol. 91. 1. Jan. 2003, pp. 178–188.

[Bie+08] Dariusz Biernacki, Jean-Louis Colaço, Grégoire Hamon, and Marc Pouzet. "Clock-directed Modular Code Generation of Synchronous Data-flow Languages." In: *ACM International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*. Tucson, Arizona, June 2008.

[Aug+14] Cédric Auger, Jean-Louis Colaço, Grégoire Hamon, and Marc Pouzet. "A Formalization and Proof of a Modular Lustre Code Generator." En préparation. 2014.

[Bou+16] Timothy Bourke, Pierre-Évariste Dagand, Marc Pouzet, and Lionel Rieg. "Verifying Clock-Directed Modular Code Generation for Lustre." En préparation. 2016.

References III

[Ler09a] Xavier Leroy. "A formally verified compiler back-end." In: *Journal of Automated Reasoning* 43.4 (2009), pp. 363–446.

[Ler09b] Xavier Leroy. "Formal verification of a realistic compiler." In: *Comms. ACM* 52.7 (2009), pp. 107–115.

[CG15] Martin Clochard and Léon Gondelman. "Double WP : Vers une preuve automatique d'un compilateur." In: *Journées Francophones des Langages Applicatifs*. INRIA. Jan. 2015.

[Chl07] Adam Chlipala. "A certified type-preserving compiler from lambda calculus to assembly language." In: *Programming Language Design and Implementation*. ACM. 2007, pp. 54–65.

[BKV09] Nick Benton, Andrew Kennedy, and Carsten Varming. "Some domain theory and denotational semantics in Coq." In: *Theorem Proving in Higher Order Logics*. volume 5674 de LNCS. 2009, pp. 115–130.

[BH09] Nick Benton and Chung-Kil Hur. "Biorthogonality, step-indexing and compiler correctness." In: *International Conference on Functional Programming*. ACM. 2009, pp. 97–108.