Verifying the Lustre modular reset

Timothy Bourke[1,2] Lélio Brun[1,2] Marc Pouzet [3,2,1]

[1]Inria Paris

[2]DI ENS

[3]UPMC

SYNCHRON'18 — November 29, 2018

Adding the modular reset to Vélus

Adding the modular reset to Vélus

A real challenge

- Semantics: several alternatives

Adding the modular reset to Vélus

A real challenge

- Semantics: several alternatives
- Compilation

Adding the modular reset to Vélus

A real challenge

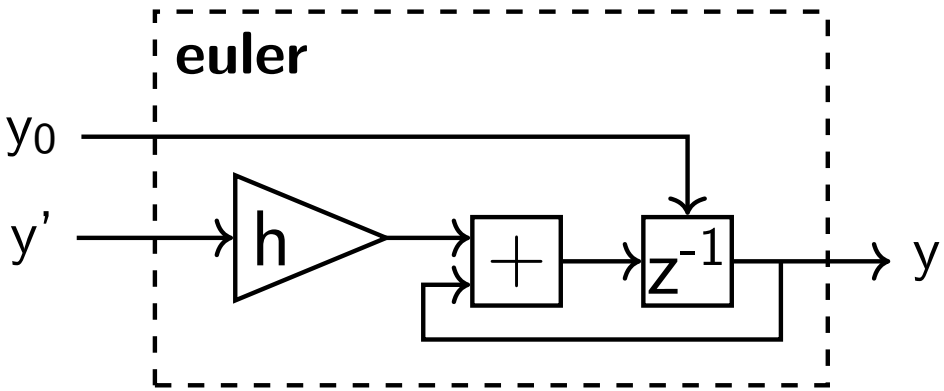
- Semantics: several alternatives
- Compilation
  - Update the existing proof: fundamental modifications

Adding the modular reset to Vélus

A real challenge

- Semantics: several alternatives
- Compilation
  - Update the existing proof: fundamental modifications
  - Optimization

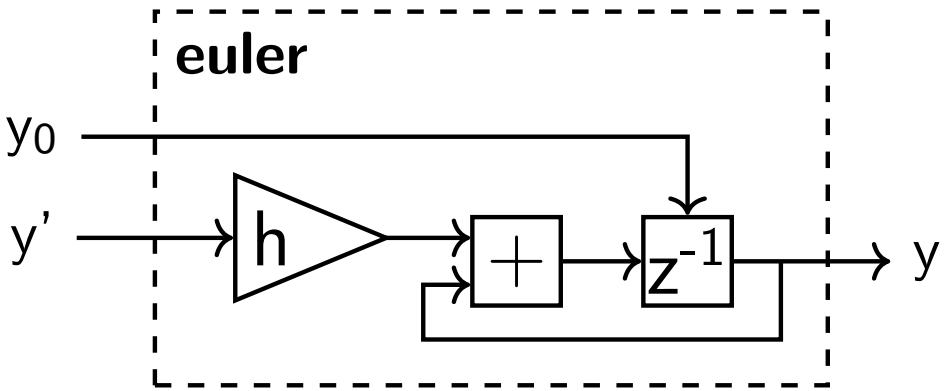
Lustre:[1] example



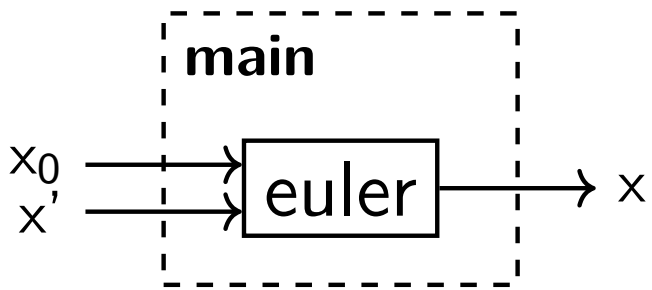

```
node euler(y0, y': real)
  returns (y: real)
  var h: real;
let
  y = y0 fby (y + y' * h);
  h = 2;
tel
```

[1] Caspi, Halbwachs, Pilaud, and Plaice (1987): "LUSTRE: A declarative language for programming synchronous systems"

Lustre:[1] example



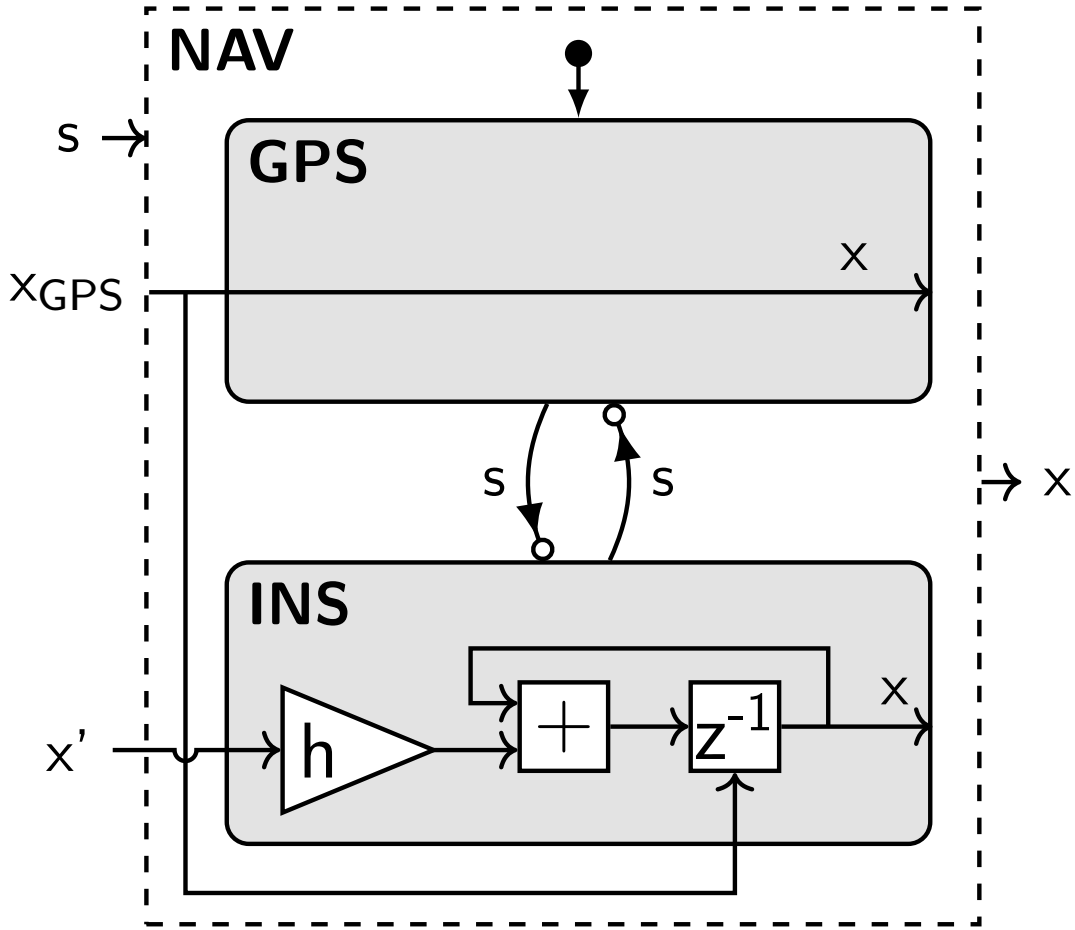

```
node euler(y0, y': real)
  returns (y: real)
  var h: real;
let
  y = y0 fby (y + y' * h);
  h = 2;
tel
```




```
node main(x0, x': real)
  returns (x: real)
let
  x = euler(x0, x');
tel
```

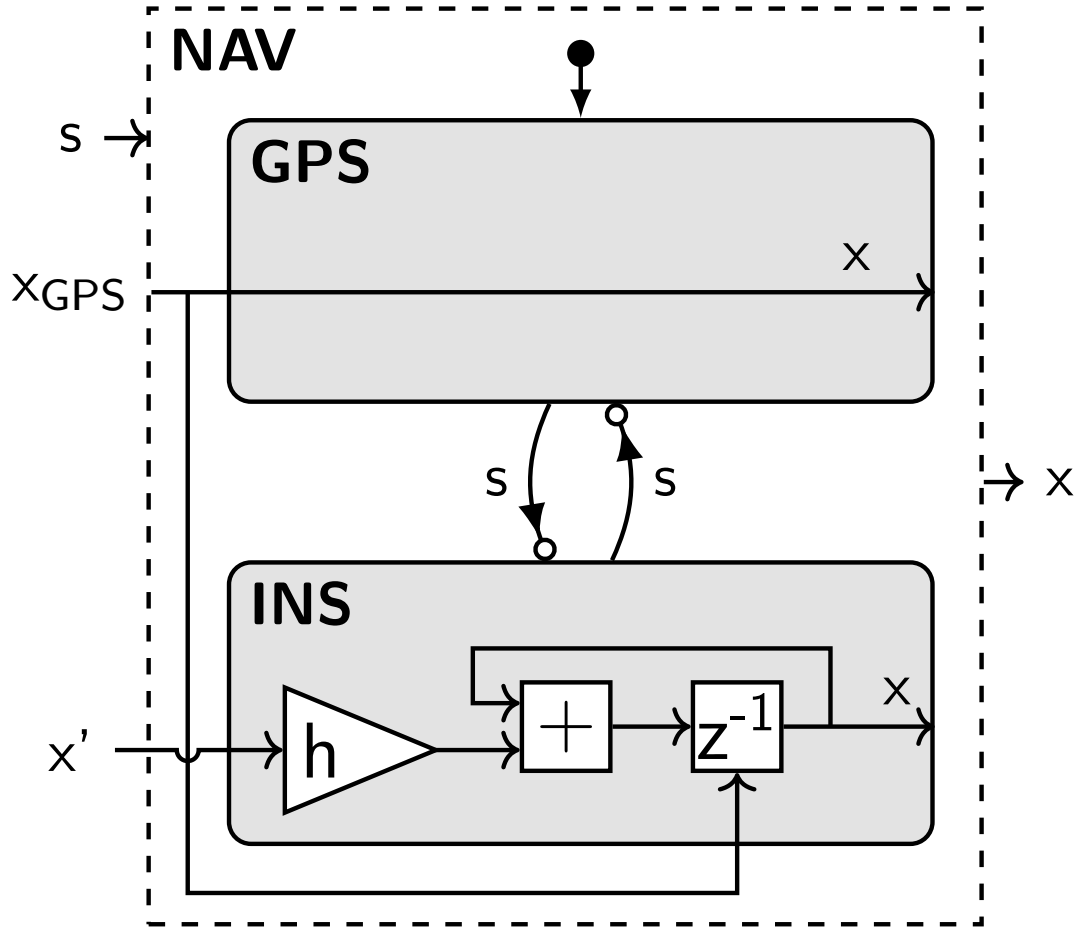
[1] Caspi, Halbwachs, Pilaud, and Plaice (1987): "LUSTRE: A declarative language for programming synchronous systems"

SCADE-like state machines and reset primitive



SCADE-like state machines and reset primitive

Can be compiled into Lustre

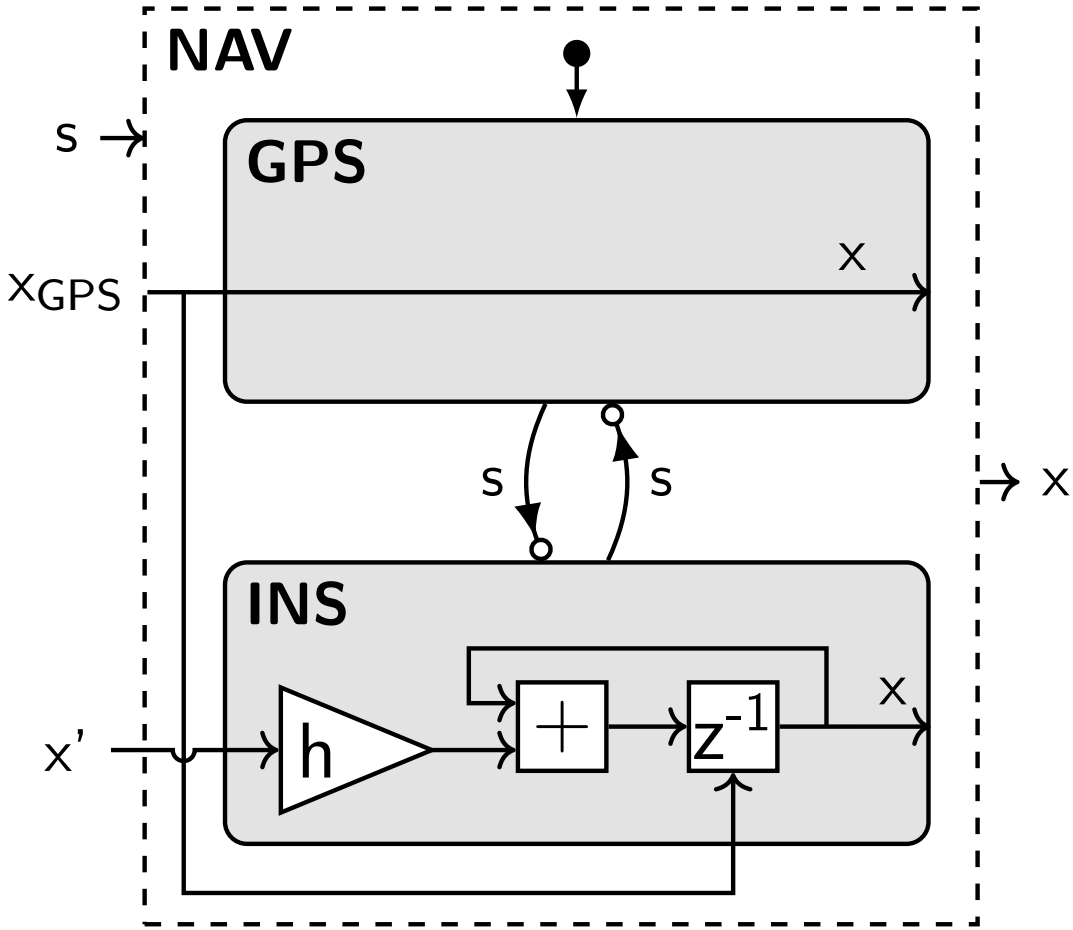


SCADE-like state machines and reset primitive

Can be compiled into Lustre

Reset:

- Reset the state of a node, ie. reinitialize the *fby*s
- Useful primitive (not only for state machines)
- How?



Without modular reset

Node application:

$f(e_1, \ldots, e_n)$

call the node f

```
node euler(y0, y': real; r: bool)
  returns (y: real)
  var h: real;
let
  y = if r then y0
      else (y0 fby (y + y' * h));
  h = 2;
tel

node main(x0, x': real)
  returns (x: real)
  var r: bool;
let
  x = euler(x0, x', r);
  r = (x' > 42);
tel
```

Without modular reset

Node application:

$f(e_1, \ldots, e_n)$

call the node f

```
node euler(y0, y': real; r: bool)
  returns (y: real)
  var h: real;
let
  y = if r then y0
      else (y0 fby (y + y' * h));
  h = 2;
tel

node main(x0, x': real)
  returns (x: real)
  var r: bool;
let
  x = euler(x0, x', r);
  r = (x' > 42);
tel
```

Without modular reset

Node application:

$f(e_1, \ldots, e_n)$

call the node f

```
node euler(y0, y': real; r: bool)
  returns (y: real)
  var h: real;
let
  y = if r then y0
      else (y0 fby (y + y' * h));
  h = 2;
tel

node main(x0, x': real)
  returns (x: real)
  var r: bool;
let
  x = euler(x0, x', r);
  r = (x' > 42);
tel
```

With modular reset

Modular reset:

$f(e_1, \ldots, e_n)$ every r

reset the internal state (delays) of f at each tick of r

```
node euler(y0, y': real)
  returns (y: real)
  var h: real;
let
  y = y0 fby (y + y' * h);
  h = 2;
tel

node main(x0, x': real)
  returns (x: real)
  var r: bool;
let
  x = euler(x0, x') every r;
  r = (x' > 42);
tel
```

Without modular reset

Node application:

$f(e_1, \ldots, e_n)$

call the node f

```
node euler(y0, y': real; r: bool)
  returns (y: real)
  var h: real;
let
  y = if r then y0
      else (y0 fby (y + y' * h));
  h = 2;
tel

node main(x0, x': real)
  returns (x: real)
  var r: bool;
let
  x = euler(x0, x', r);
  r = (x' > 42);
tel
```

With modular reset

Modular reset:

$f(e_1, \ldots, e_n)$ every r

reset the internal state (delays) of f at each tick of r

```
node euler(y0, y': real)
  returns (y: real)
  var h: real;
let
  y = y0 fby (y + y' * h);
  h = 2;
tel

node main(x0, x': real)
  returns (x: real)
  var r: bool;
let
  x = euler(x0, x') every r;
  r = (x' > 42);
tel
```

A simpler example

```
node nat(i: int) returns (n: int)
let
  n = i fby (n + 1);
tel
```

A simpler example

```
node nat(i: int) returns (n: int)
let
  n = i fby (n + 1);
tel
```

r	F
i	0
$nat(i)$	0
$nat(i)$ every r	0

A simpler example

```
node nat(i: int) returns (n: int)
let
  n = i fby (n + 1);
tel
```

r	F	F
i	0	5
$nat(i)$	0	1
$nat(i)$ every r	0	1

A simpler example

```
node nat(i: int) returns (n: int)
let
  n = i fby (n + 1);
tel
```

r	F	F	T
i	0	5	10
$nat(i)$	0	1	2
$nat(i)$ every r	0	1	10

A simpler example

```
node nat(i: int) returns (n: int)
let
  n = i fby (n + 1);
tel
```

r	F	F	T	F
i	0	5	10	15
$nat(i)$	0	1	2	3
$nat(i)$ every r	0	1	10	11

A simpler example

```
node nat(i: int) returns (n: int)
let
  n = i fby (n + 1);
tel
```

r	F	F	T	F	F
i	0	5	10	15	20
$nat(i)$	0	1	2	3	4
$nat(i)$ every r	0	1	10	11	12

A simpler example

```
node nat(i: int) returns (n: int)
let
  n = i fby (n + 1);
tel
```

r	F	F	T	F	F	T
i	0	5	10	15	20	25
$nat(i)$	0	1	2	3	4	5
$nat(i)$ every r	0	1	10	11	12	25

A simpler example

```
node nat(i: int) returns (n: int)
let
  n = i fby (n + 1);
tel
```

r	F	F	T	F	F	T	F
i	0	5	10	15	20	25	30
$nat(i)$	0	1	2	3	4	5	6
$nat(i)$ every r	0	1	10	11	12	25	26

A simpler example

```
node nat(i: int) returns (n: int)
let
  n = i fby (n + 1);
tel
```

r	F	F	T	F	F	T	F	T
i	0	5	10	15	20	25	30	35
$nat(i)$	0	1	2	3	4	5	6	7
$nat(i)$ every r	0	1	10	11	12	25	26	35

A simpler example

```
node nat(i: int) returns (n: int)
let
  n = i fby (n + 1);
tel
```

r	F	F	T	F	F	T	F	T	F
i	0	5	10	15	20	25	30	35	40
$nat(i)$	0	1	2	3	4	5	6	7	8
$nat(i)$ every r	0	1	10	11	12	25	26	35	36

A simpler example

```
node nat(i: int) returns (n: int)
let
  n = i fby (n + 1);
tel
```

r	F	F	T	F	F	T	F	T	F	$\cdots$
i	0	5	10	15	20	25	30	35	40	$\cdots$
$nat(i)$	0	1	2	3	4	5	6	7	8	$\cdots$
$nat(i)$ every r	0	1	10	11	12	25	26	35	36	$\cdots$

Semantics?

A recursive intuition, not a valid definition in Lustre:[2]

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

node reset_f(x: int, r: bool) returns (y: int)
  var c: bool;
let
  c = true_until(r);
  y = merge c (f(x when c))
              (reset_f((x, r) whenot c));
tel
```

r	F	F	F	T	F	T	F	$\cdots$
c	T	T	T	F	F	F	F	$\cdots$

[2] 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	$\cdots$
i	0	5	10	15	20	25	30	35	40	$\cdots$
$nat(i)$ every r	0	1	10	11	12	25	26	35	36	$\cdots$

Infinitely unrolling the recursion

r	F	F	T	F	F	T	F	T	F	$\cdots$
count r	0	0	1	1	1	2	2	3	3	$\cdots$
i	0	5	10	15	20	25	30	35	40	$\cdots$
$nat(i)$ every r	0	1	10	11	12	25	26	35	36	$\cdots$

Infinitely unrolling the recursion

r	F	F	T	F	F	T	F	T	F	$\cdots$
count r	0	0	1	1	1	2	2	3	3	$\cdots$
i	0	5	10	15	20	25	30	35	40	$\cdots$
mask 0 r i	0	5	–	–	–	–	–	–	–	$\cdots$
$nat(i)$ every r	0	1	10	11	12	25	26	35	36	$\cdots$

Infinitely unrolling the recursion

r	F	F	T	F	F	T	F	T	F	$\cdots$
count r	0	0	1	1	1	2	2	3	3	$\cdots$
i	0	5	10	15	20	25	30	35	40	$\cdots$
mask 0 r i	0	5	–	–	–	–	–	–	–	$\cdots$
nat(mask 0 r i)	0	1	–	–	–	–	–	–	–	$\cdots$
$nat(i)$ every r	0	1	10	11	12	25	26	35	36	$\cdots$

Infinitely unrolling the recursion

r	F	F	T	F	F	T	F	T	F	$\cdots$
count r	0	0	1	1	1	2	2	3	3	$\cdots$
i	0	5	10	15	20	25	30	35	40	$\cdots$
mask 0 r i	0	5	–	–	–	–	–	–	–	$\cdots$
nat(mask 0 r i)	0	1	–	–	–	–	–	–	–	$\cdots$
mask 1 r i	–	–	10	15	20	–	–	–	–	$\cdots$
$nat(i)$ every r	0	1	10	11	12	25	26	35	36	$\cdots$

Infinitely unrolling the recursion

r	F	F	T	F	F	T	F	T	F	$\cdots$
count r	0	0	1	1	1	2	2	3	3	$\cdots$
i	0	5	10	15	20	25	30	35	40	$\cdots$
mask 0 r i	0	5	–	–	–	–	–	–	–	$\cdots$
nat(mask 0 r i)	0	1	–	–	–	–	–	–	–	$\cdots$
mask 1 r i	–	–	10	15	20	–	–	–	–	$\cdots$
nat(mask 1 r i)	–	–	10	11	12	–	–	–	–	$\cdots$
$nat(i)$ every r	0	1	10	11	12	25	26	35	36	$\cdots$

Infinitely unrolling the recursion

r	F	F	T	F	F	T	F	T	F	$\cdots$
count r	0	0	1	1	1	2	2	3	3	$\cdots$
i	0	5	10	15	20	25	30	35	40	$\cdots$
mask 0 r i	0	5	–	–	–	–	–	–	–	$\cdots$
nat(mask 0 r i)	0	1	–	–	–	–	–	–	–	$\cdots$
mask 1 r i	–	–	10	15	20	–	–	–	–	$\cdots$
nat(mask 1 r i)	–	–	10	11	12	–	–	–	–	$\cdots$
mask 2 r i	–	–	–	–	–	25	30	–	–	$\cdots$
$nat(i)$ every r	0	1	10	11	12	25	26	35	36	$\cdots$

Infinitely unrolling the recursion

r	F	F	T	F	F	T	F	T	F	$\cdots$
count r	0	0	1	1	1	2	2	3	3	$\cdots$
i	0	5	10	15	20	25	30	35	40	$\cdots$
mask 0 r i	0	5	–	–	–	–	–	–	–	$\cdots$
nat(mask 0 r i)	0	1	–	–	–	–	–	–	–	$\cdots$
mask 1 r i	–	–	10	15	20	–	–	–	–	$\cdots$
nat(mask 1 r i)	–	–	10	11	12	–	–	–	–	$\cdots$
mask 2 r i	–	–	–	–	–	25	30	–	–	$\cdots$
nat(mask 2 r i)	–	–	–	–	–	25	26	–	–	$\cdots$
$nat(i)$ every r	0	1	10	11	12	25	26	35	36	$\cdots$

Infinitely unrolling the recursion

r	F	F	T	F	F	T	F	T	F	$\cdots$
count r	0	0	1	1	1	2	2	3	3	$\cdots$
i	0	5	10	15	20	25	30	35	40	$\cdots$
mask 0 r i	0	5	–	–	–	–	–	–	–	$\cdots$
nat(mask 0 r i)	0	1	–	–	–	–	–	–	–	$\cdots$
mask 1 r i	–	–	10	15	20	–	–	–	–	$\cdots$
nat(mask 1 r i)	–	–	10	11	12	–	–	–	–	$\cdots$
mask 2 r i	–	–	–	–	–	25	30	–	–	$\cdots$
nat(mask 2 r i)	–	–	–	–	–	25	26	–	–	$\cdots$
mask 3 r i	–	–	–	–	–	–	–	35	40	$\cdots$
$nat(i)$ every r	0	1	10	11	12	25	26	35	36	$\cdots$

Infinitely unrolling the recursion

r	F	F	T	F	F	T	F	T	F	$\cdots$
count r	0	0	1	1	1	2	2	3	3	$\cdots$
i	0	5	10	15	20	25	30	35	40	$\cdots$
mask 0 r i	0	5	–	–	–	–	–	–	–	$\cdots$
nat(mask 0 r i)	0	1	–	–	–	–	–	–	–	$\cdots$
mask 1 r i	–	–	10	15	20	–	–	–	–	$\cdots$
nat(mask 1 r i)	–	–	10	11	12	–	–	–	–	$\cdots$
mask 2 r i	–	–	–	–	–	25	30	–	–	$\cdots$
nat(mask 2 r i)	–	–	–	–	–	25	26	–	–	$\cdots$
mask 3 r i	–	–	–	–	–	–	–	35	40	$\cdots$
nat(mask 3 r i)	–	–	–	–	–	–	–	35	36	$\cdots$
$nat(i)$ every r	0	1	10	11	12	25	26	35	36	$\cdots$

Infinitely unrolling the recursion

r	F	F	T	F	F	T	F	T	F	$\cdots$
count r	0	0	1	1	1	2	2	3	3	$\cdots$
i	0	5	10	15	20	25	30	35	40	$\cdots$
mask 0 r i	0	5	–	–	–	–	–	–	–	$\cdots$
nat(mask 0 r i)	0	1	–	–	–	–	–	–	–	$\cdots$
mask 1 r i	–	–	10	15	20	–	–	–	–	$\cdots$
nat(mask 1 r i)	–	–	10	11	12	–	–	–	–	$\cdots$
mask 2 r i	–	–	–	–	–	25	30	–	–	$\cdots$
nat(mask 2 r i)	–	–	–	–	–	25	26	–	–	$\cdots$
mask 3 r i	–	–	–	–	–	–	–	35	40	$\cdots$
nat(mask 3 r i)	–	–	–	–	–	–	–	35	36	$\cdots$
⋮										
$nat(i)$ every r	0	1	10	11	12	25	26	35	36	$\cdots$

Formal semantics

Node application

$$\frac{}{\vdash_{\mathsf{eqn}} \vec{x} = f(\vec{e})}$$

Formal semantics

Node application

$$\frac{\vdash_{\mathsf{exp}} \vec{e} \Downarrow \vec{es}}{\vdash_{\mathsf{eqn}} \vec{x} = f(\vec{e})}$$

Formal semantics

Node application

$$\frac{\vdash_{\mathsf{exp}} \vec{e} \Downarrow \overrightarrow{es} \qquad \vdash_{\mathsf{node}} f(\overrightarrow{es}) \Downarrow \overrightarrow{xs}}{\vdash_{\mathsf{eqn}} \vec{x} = f(\vec{e})}$$

Formal semantics

Node application

$$\frac{\vdash_{\mathsf{exp}} \vec{e} \Downarrow \vec{es} \qquad \vdash_{\mathsf{node}} f(\vec{es}) \Downarrow \vec{xs} \qquad \vdash_{\mathsf{var}} \vec{x} \Downarrow \vec{xs}}{\vdash_{\mathsf{eqn}} \vec{x} = f(\vec{e})}$$

Formal semantics

Node application

$$\frac{\vdash_{\mathsf{exp}} \vec{e} \Downarrow \vec{es} \qquad \vdash_{\mathsf{node}} f(\vec{es}) \Downarrow \vec{xs} \qquad \vdash_{\mathsf{var}} \vec{x} \Downarrow \vec{xs}}{\vdash_{\mathsf{eqn}} \vec{x} = f(\vec{e})}$$

Modular reset

$$\frac{}{\vdash_{\mathsf{eqn}} \vec{x} = f(\vec{e}) \text{ every } r}$$

Formal semantics

Node application

$$\frac{\vdash_{\mathsf{exp}} \vec{e} \Downarrow \overrightarrow{es} \qquad \vdash_{\mathsf{node}} f(\overrightarrow{es}) \Downarrow \overrightarrow{xs} \qquad \vdash_{\mathsf{var}} \vec{x} \Downarrow \overrightarrow{xs}}{\vdash_{\mathsf{eqn}} \vec{x} = f(\vec{e})}$$

Modular reset

$$\frac{\vdash_{\mathsf{exp}} \vec{e} \Downarrow \overrightarrow{es} \qquad\qquad \vdash_{\mathsf{var}} \vec{x} \Downarrow \overrightarrow{xs}}{\vdash_{\mathsf{eqn}} \vec{x} = f(\vec{e}) \text{ every } r}$$

Formal semantics

Node application

$$\frac{\vdash_{\mathsf{exp}} \vec{e} \Downarrow \vec{es} \qquad \vdash_{\mathsf{node}} f(\vec{es}) \Downarrow \vec{xs} \qquad \vdash_{\mathsf{var}} \vec{x} \Downarrow \vec{xs}}{\vdash_{\mathsf{eqn}} \vec{x} = f(\vec{e})}$$

Modular reset

$$\frac{\begin{array}{c}\vdash_{\mathsf{var}} r \Downarrow rs \\ \vdash_{\mathsf{exp}} \vec{e} \Downarrow \vec{es} \end{array} \qquad \vdash_{\mathsf{var}} \vec{x} \Downarrow \vec{xs}}{\vdash_{\mathsf{eqn}} \vec{x} = f(\vec{e}) \text{ every } r}$$

Formal semantics

Node application

$$\frac{\vdash_{\mathsf{exp}} \vec{e} \Downarrow \vec{es} \qquad \vdash_{\mathsf{node}} f(\vec{es}) \Downarrow \vec{xs} \qquad \vdash_{\mathsf{var}} \vec{x} \Downarrow \vec{xs}}{\vdash_{\mathsf{eqn}} \vec{x} = f(\vec{e})}$$

Modular reset

$$\frac{\vdash_{\mathsf{var}} r \Downarrow rs \qquad rk = \mathsf{boolof}\ rs \qquad \vdash_{\mathsf{exp}} \vec{e} \Downarrow \vec{es} \qquad \vdash_{\mathsf{var}} \vec{x} \Downarrow \vec{xs}}{\vdash_{\mathsf{eqn}} \vec{x} = f(\vec{e})\ \mathsf{every}\ r}$$

Formal semantics

Node application

$$\frac{\vdash_{\mathsf{exp}} \vec{e} \Downarrow \vec{es} \qquad \vdash_{\mathsf{node}} f(\vec{es}) \Downarrow \vec{xs} \qquad \vdash_{\mathsf{var}} \vec{x} \Downarrow \vec{xs}}{\vdash_{\mathsf{eqn}} \vec{x} = f(\vec{e})}$$

Modular reset

$$\frac{\vdash_{\mathsf{var}} r \Downarrow rs \qquad rk = \mathsf{boolof}\ rs \qquad \vdash_{\mathsf{exp}} \vec{e} \Downarrow \vec{es} \qquad rk \vdash_{\mathsf{reset}} f(\vec{es}) \Downarrow \vec{xs} \qquad \vdash_{\mathsf{var}} \vec{x} \Downarrow \vec{xs}}{\vdash_{\mathsf{eqn}} \vec{x} = f(\vec{e})\ \mathsf{every}\ r}$$

Formal semantics

Node application

$$\frac{\vdash_{\text{exp}} \vec{e} \Downarrow \vec{es} \qquad \vdash_{\text{node}} f(\vec{es}) \Downarrow \vec{xs} \qquad \vdash_{\text{var}} \vec{x} \Downarrow \vec{xs}}{\vdash_{\text{eqn}} \vec{x} = f(\vec{e})}$$

Modular reset

$$\frac{\begin{array}{c} \vdash_{\text{var}} r \Downarrow rs \qquad rk = \text{boolof } rs \\ \vdash_{\text{exp}} \vec{e} \Downarrow \vec{es} \qquad rk \vdash_{\text{reset}} f(\vec{es}) \Downarrow \vec{xs} \qquad \vdash_{\text{var}} \vec{x} \Downarrow \vec{xs} \end{array}}{\vdash_{\text{eqn}} \vec{x} = f(\vec{e}) \text{ every } r}$$

$$\frac{}{rk \vdash_{\text{reset}} f(\vec{xs}) \Downarrow \vec{ys}}$$

Formal semantics

Node application

$$\frac{\vdash_{\text{exp}} \vec{e} \Downarrow \overrightarrow{es} \qquad \vdash_{\text{node}} f(\overrightarrow{es}) \Downarrow \overrightarrow{xs} \qquad \vdash_{\text{var}} \vec{x} \Downarrow \overrightarrow{xs}}{\vdash_{\text{eqn}} \vec{x} = f(\vec{e})}$$

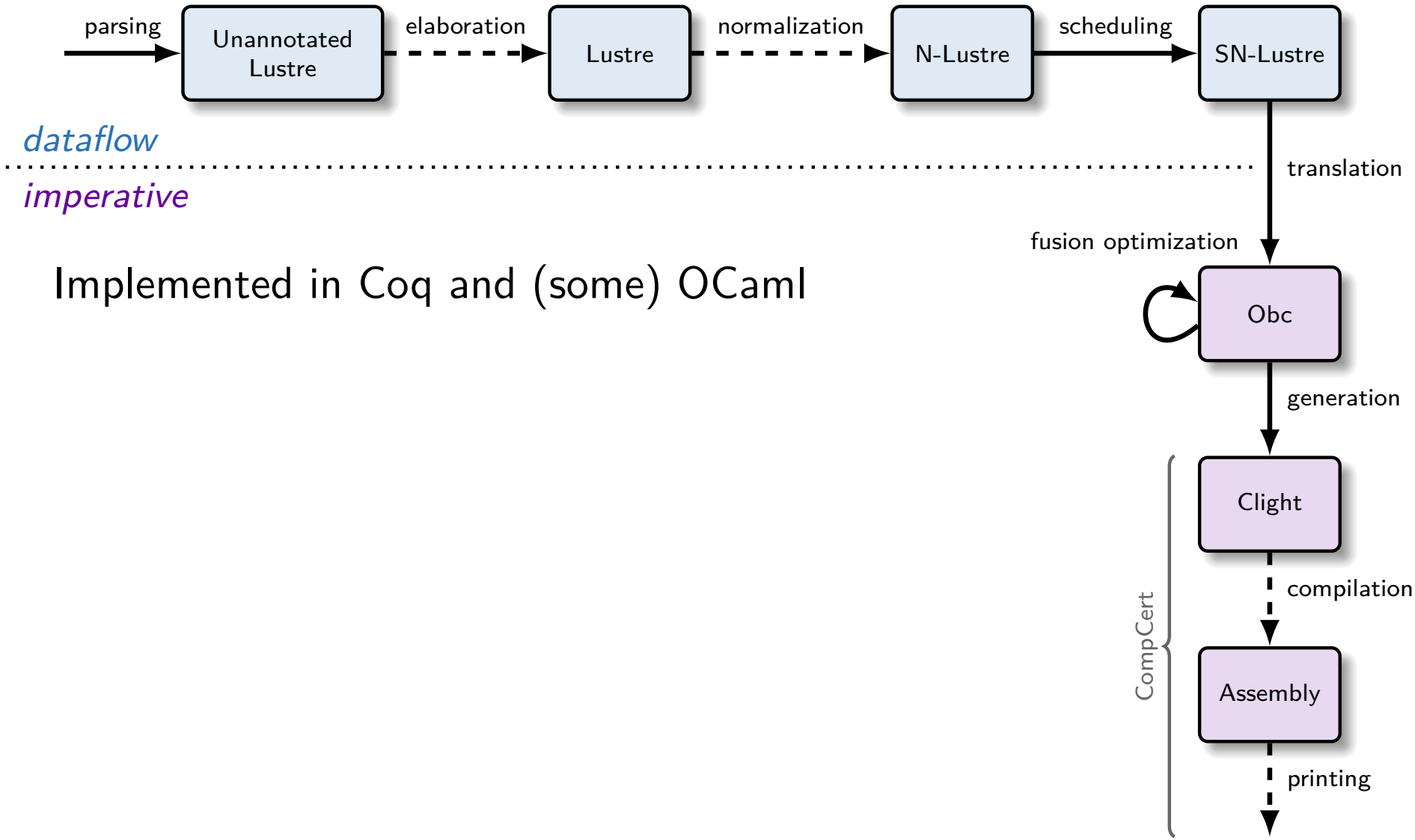
Modular reset

$$\frac{\begin{array}{c}\vdash_{\text{var}} r \Downarrow rs \qquad rk = \text{boolof } rs \\ \vdash_{\text{exp}} \vec{e} \Downarrow \overrightarrow{es} \qquad rk \vdash_{\text{reset}} f(\overrightarrow{es}) \Downarrow \overrightarrow{xs} \qquad \vdash_{\text{var}} \vec{x} \Downarrow \overrightarrow{xs}\end{array}}{\vdash_{\text{eqn}} \vec{x} = f(\vec{e}) \text{ every } r}$$

Use of an universally quantified relation as a constraint:

$$\frac{\forall k,\ \vdash_{\text{node}} f(\text{mask } k\ rk\ \overrightarrow{xs}) \Downarrow \text{mask } k\ rk\ \overrightarrow{ys}}{rk \vdash_{\text{reset}} f(\overrightarrow{xs}) \Downarrow \overrightarrow{ys}}$$

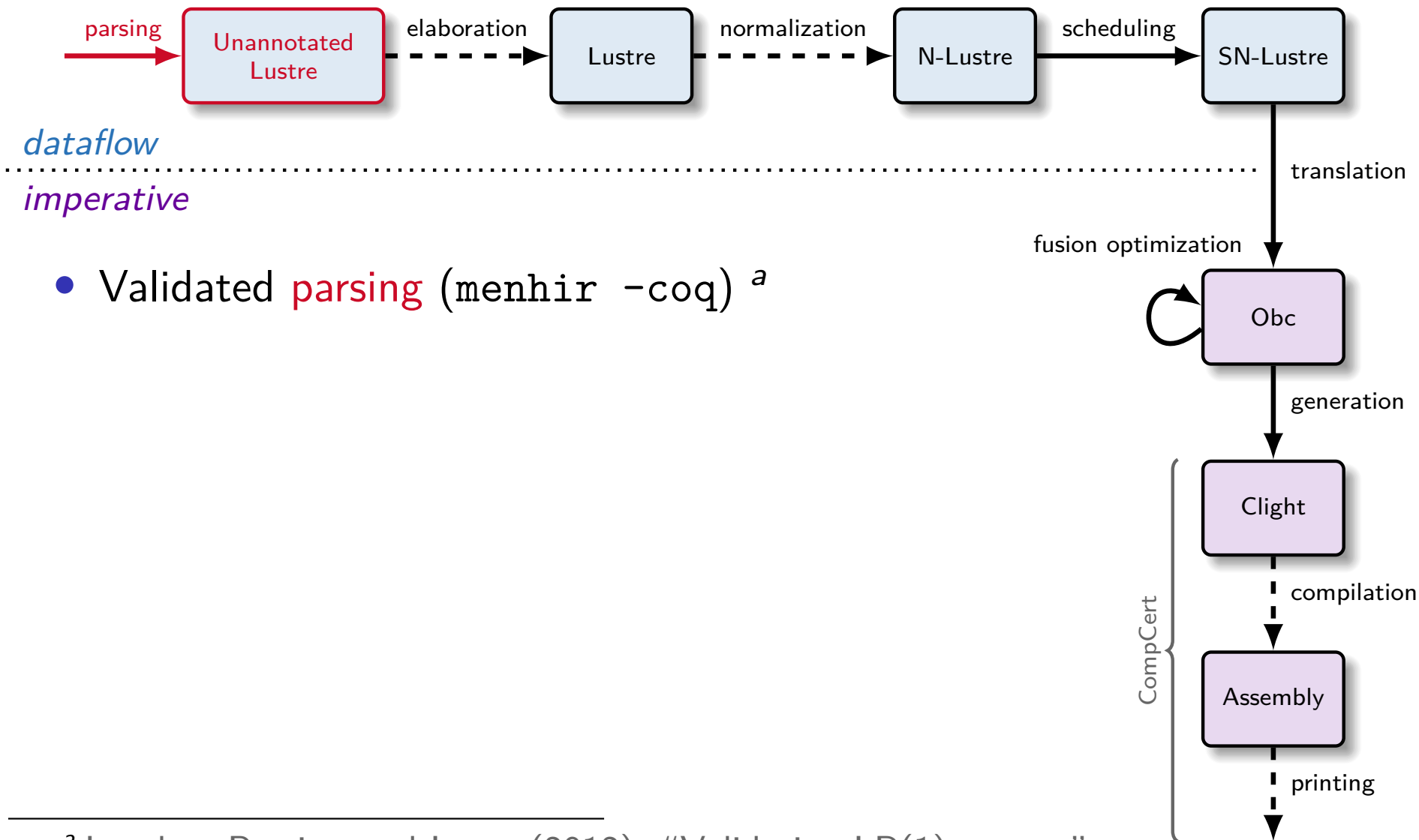
Vélus:[3] a verified compiler



Implemented in Coq and (some) OCaml

[3] Bourke, Brun, Dagand, Leroy, Pouzet, and Rieg (2017): "A Formally Verified Compiler for Lustre"

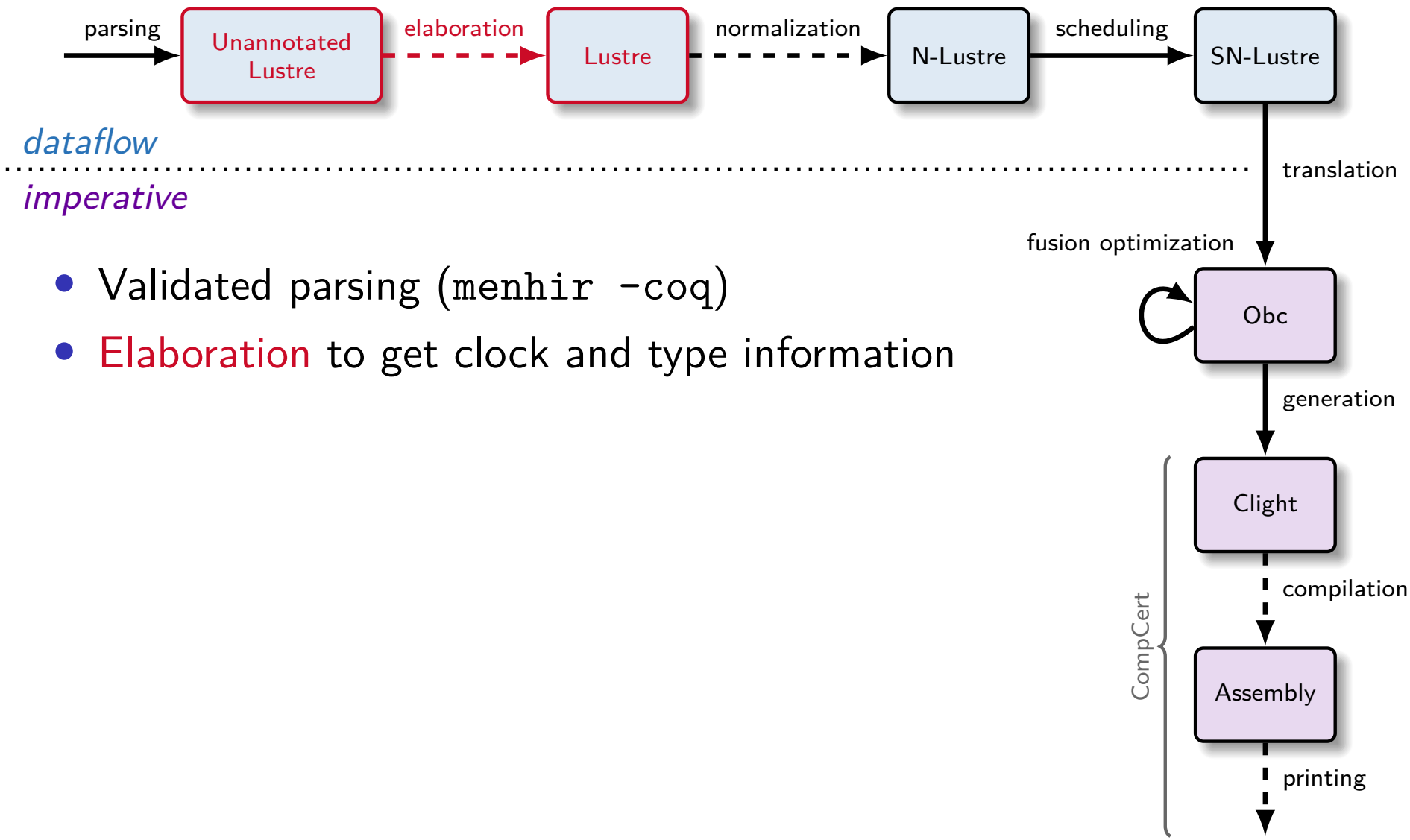
Vélus: a verified compiler



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

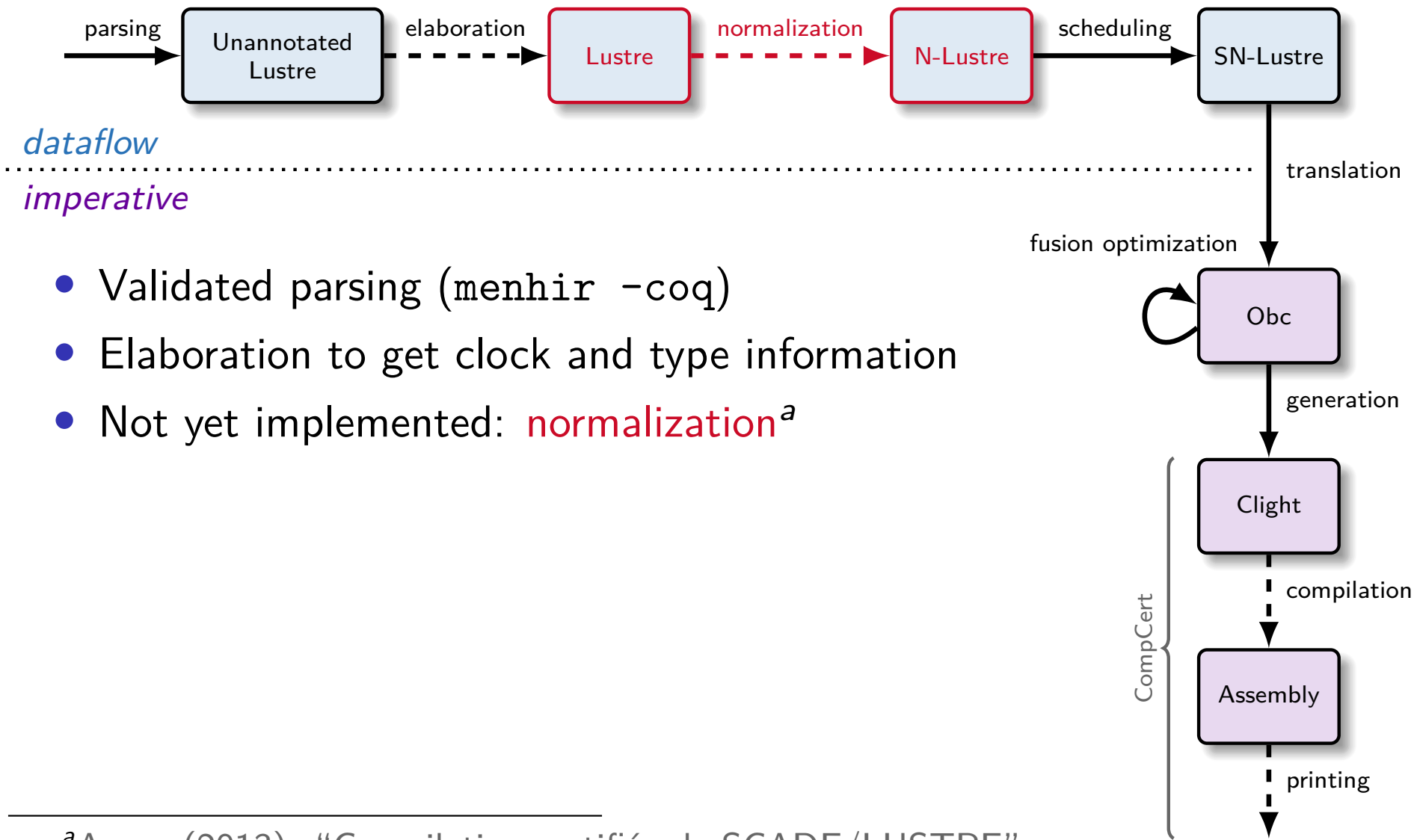
[a] Jourdan, Pottier, and Leroy (2012): "Validating LR(1) parsers"

Vélus: a verified compiler



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

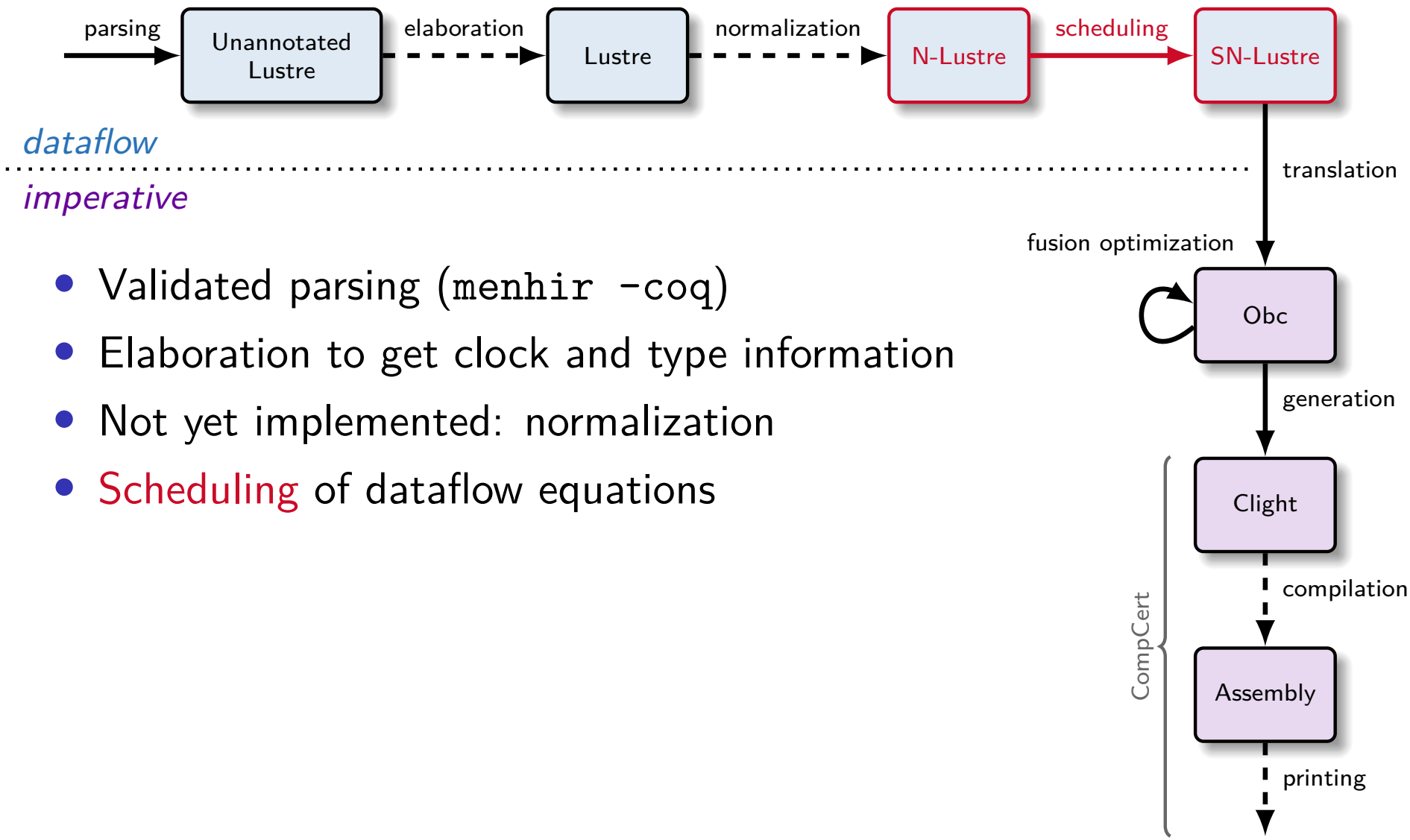
Vélus: a verified compiler



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

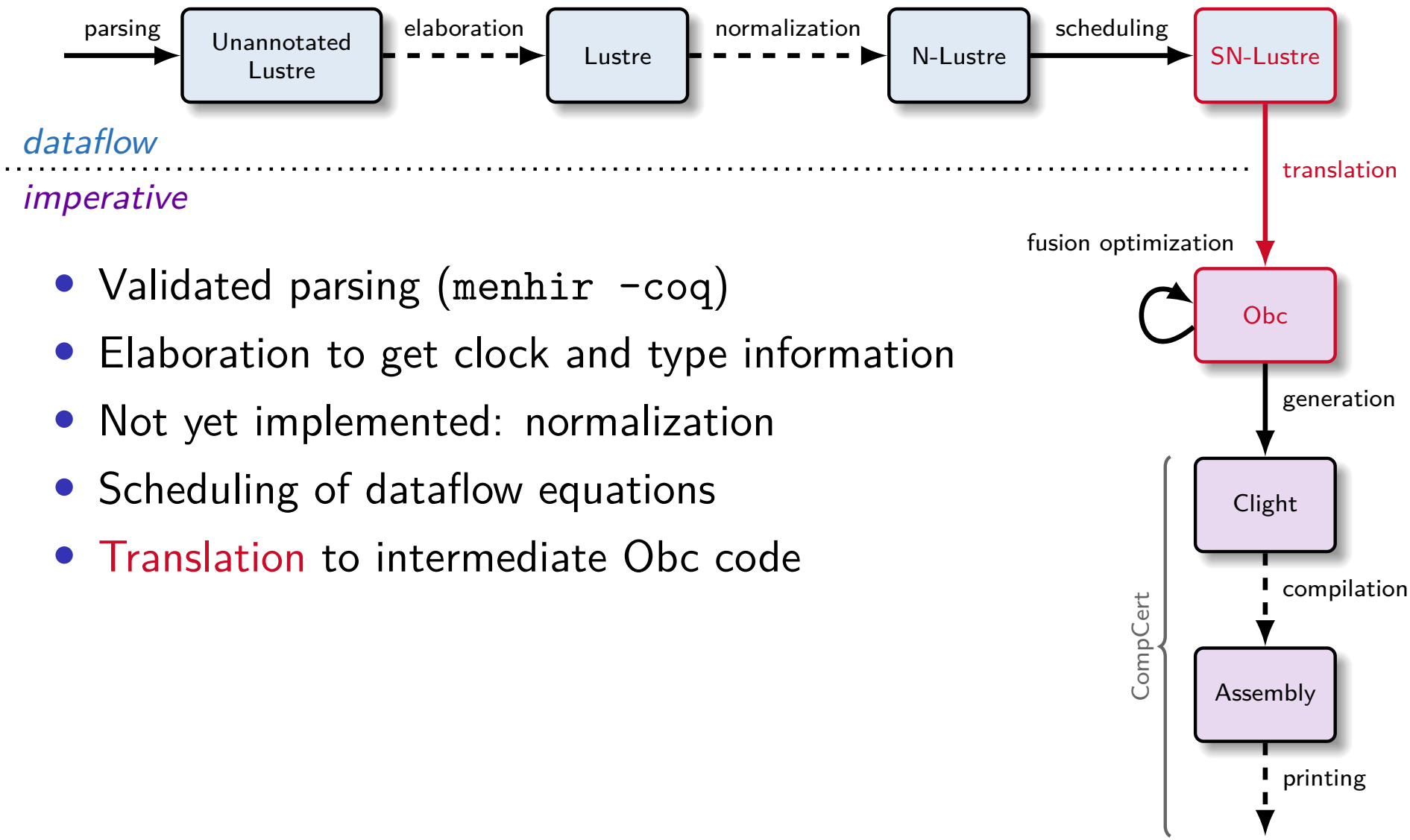
[a] Auger (2013): "Compilation certifiée de SCADE/LUSTRE"

Vélus: a verified compiler



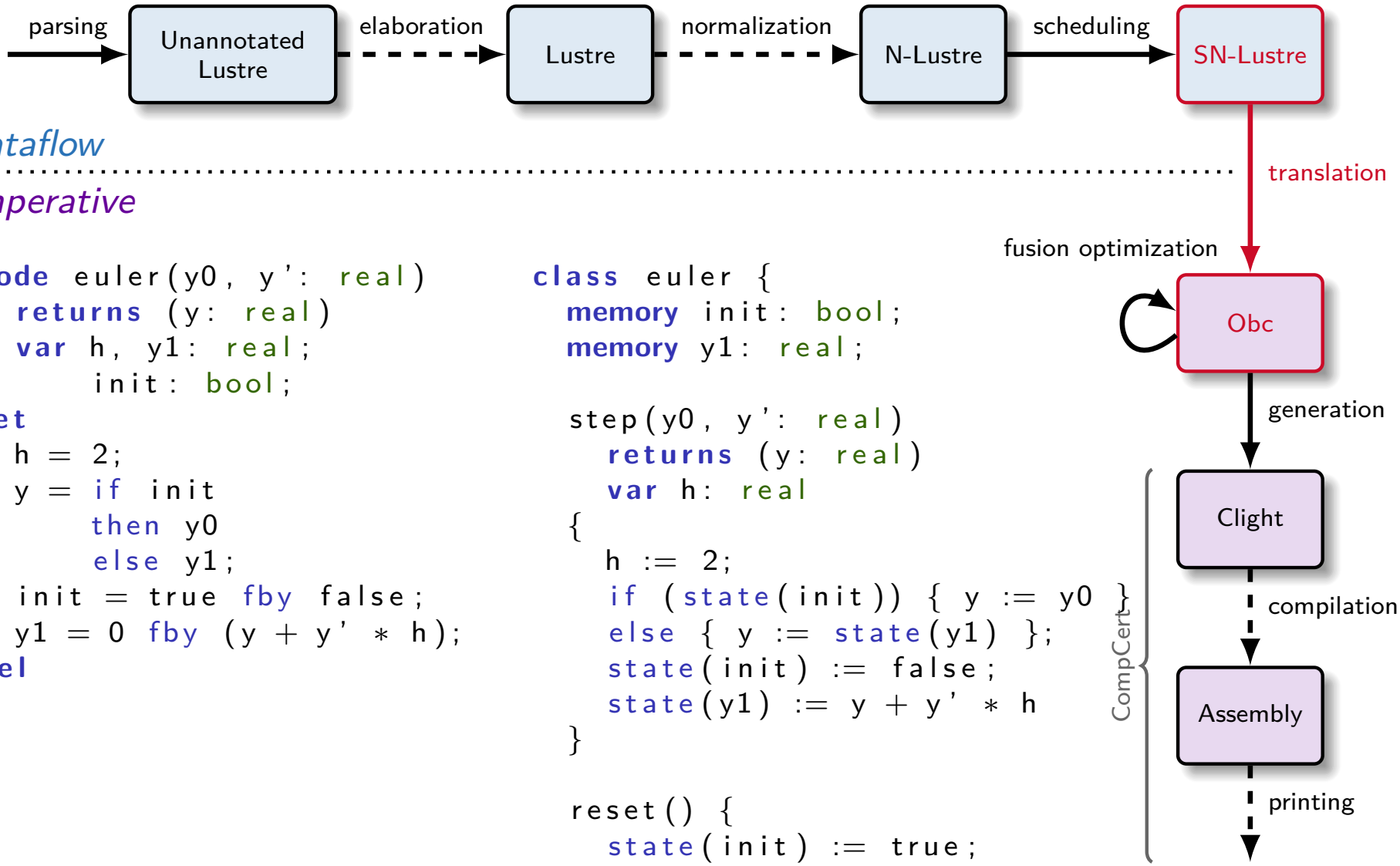
- Validated parsing (`menhir -coq`)
- Elaboration to get clock and type information
- Not yet implemented: normalization
- Scheduling of dataflow equations

Vélus: a verified compiler



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

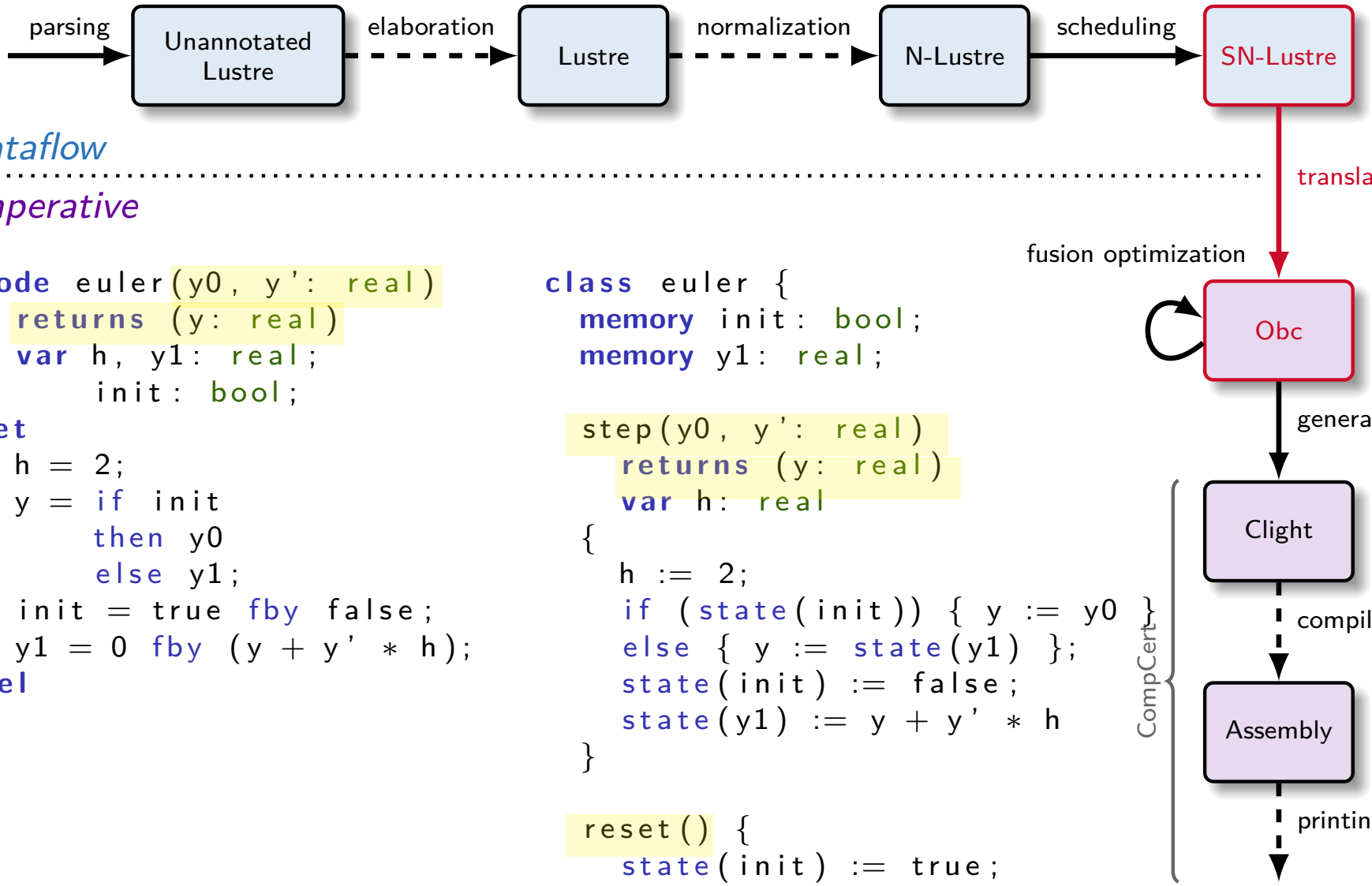
Vélus: a verified compiler




```
node euler(y0, y': real)
  returns (y: real)
  var h, y1: real;
      init: bool;
let
  h = 2;
  y = if init
      then y0
      else y1;
  init = true fby false;
  y1 = 0 fby (y + y' * h);
tel
```

```
class euler {
  memory init: bool;
  memory y1: real;

  step(y0, y': real)
    returns (y: real)
    var h: real
  {
    h := 2;
    if (state(init)) { y := y0 }
    else { y := state(y1) };
    state(init) := false;
    state(y1) := y + y' * h
  }

  reset() {
    state(init) := true;
    state(y1) := 0
  }
}
```

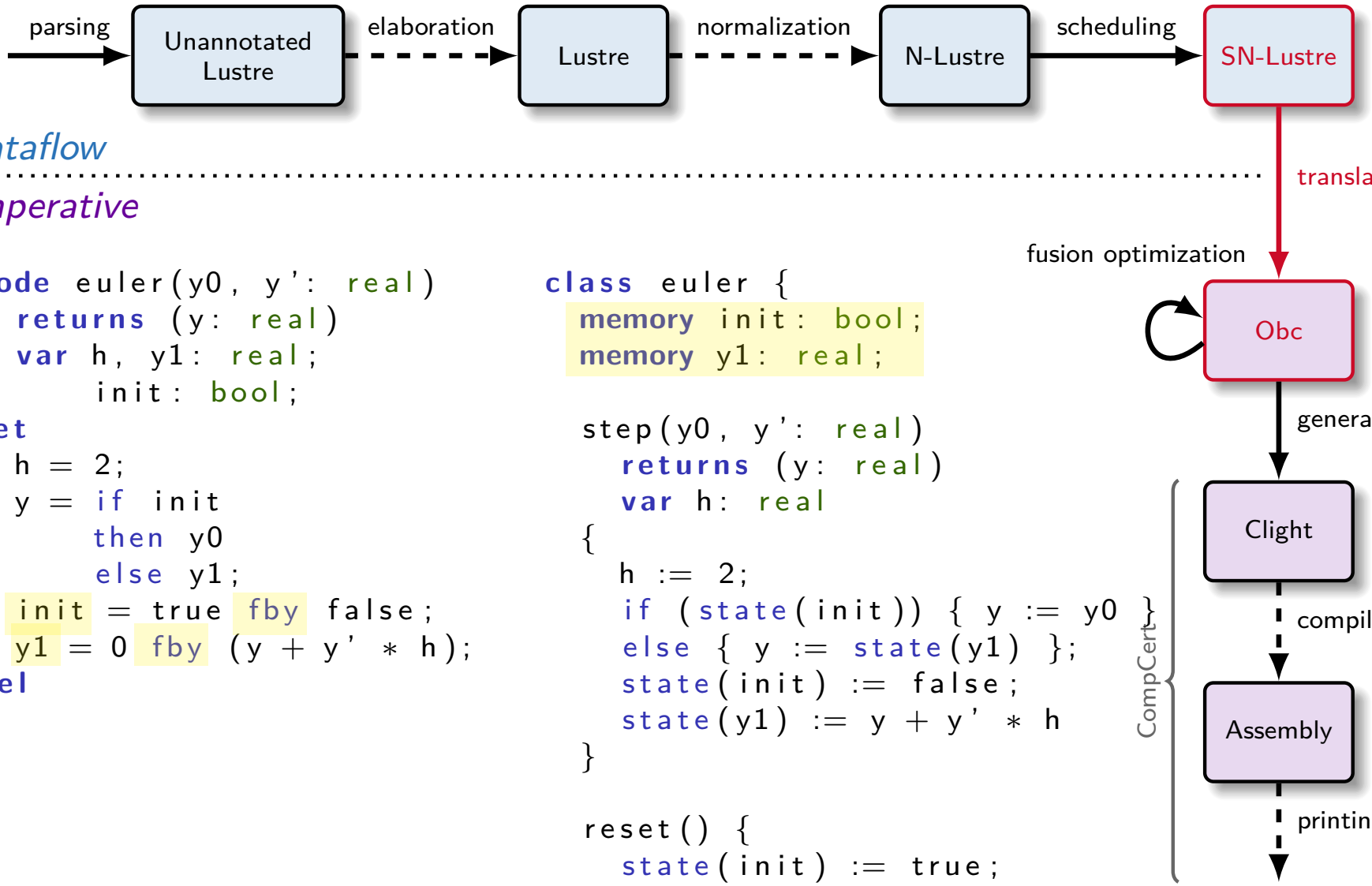
Vélus: a verified compiler




```
node euler(y0, y': real)
  returns (y: real)
  var h, y1: real;
      init: bool;
let
  h = 2;
  y = if init
      then y0
      else y1;
  init = true fby false;
  y1 = 0 fby (y + y' * h);
tel
```

```
class euler {
  memory init: bool;
  memory y1: real;

  step(y0, y': real)
    returns (y: real)
    var h: real
  {
    h := 2;
    if (state(init)) { y := y0 }
    else { y := state(y1) };
    state(init) := false;
    state(y1) := y + y' * h
  }

  reset() {
    state(init) := true;
    state(y1) := 0
  }
}
```

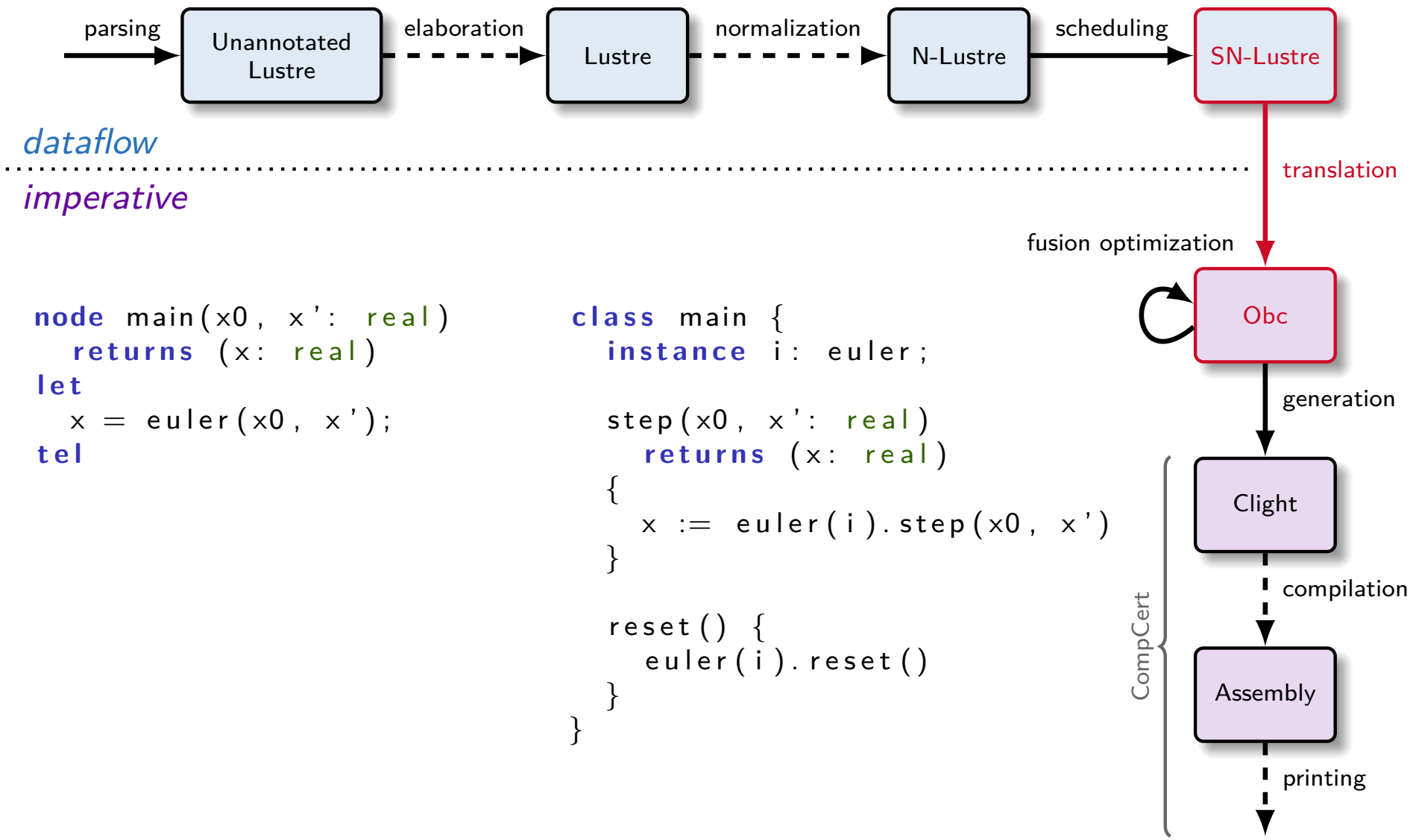
Vélus: a verified compiler




```
node euler(y0, y': real)
  returns (y: real)
  var h, y1: real;
      init: bool;
let
  h = 2;
  y = if init
      then y0
      else y1;
  init = true fby false;
  y1 = 0 fby (y + y' * h);
tel
```

```
class euler {
  memory init: bool;
  memory y1: real;

  step(y0, y': real)
    returns (y: real)
    var h: real
  {
    h := 2;
    if (state(init)) { y := y0 }
    else { y := state(y1) };
    state(init) := false;
    state(y1) := y + y' * h
  }

  reset() {
    state(init) := true;
    state(y1) := 0
  }
}
```

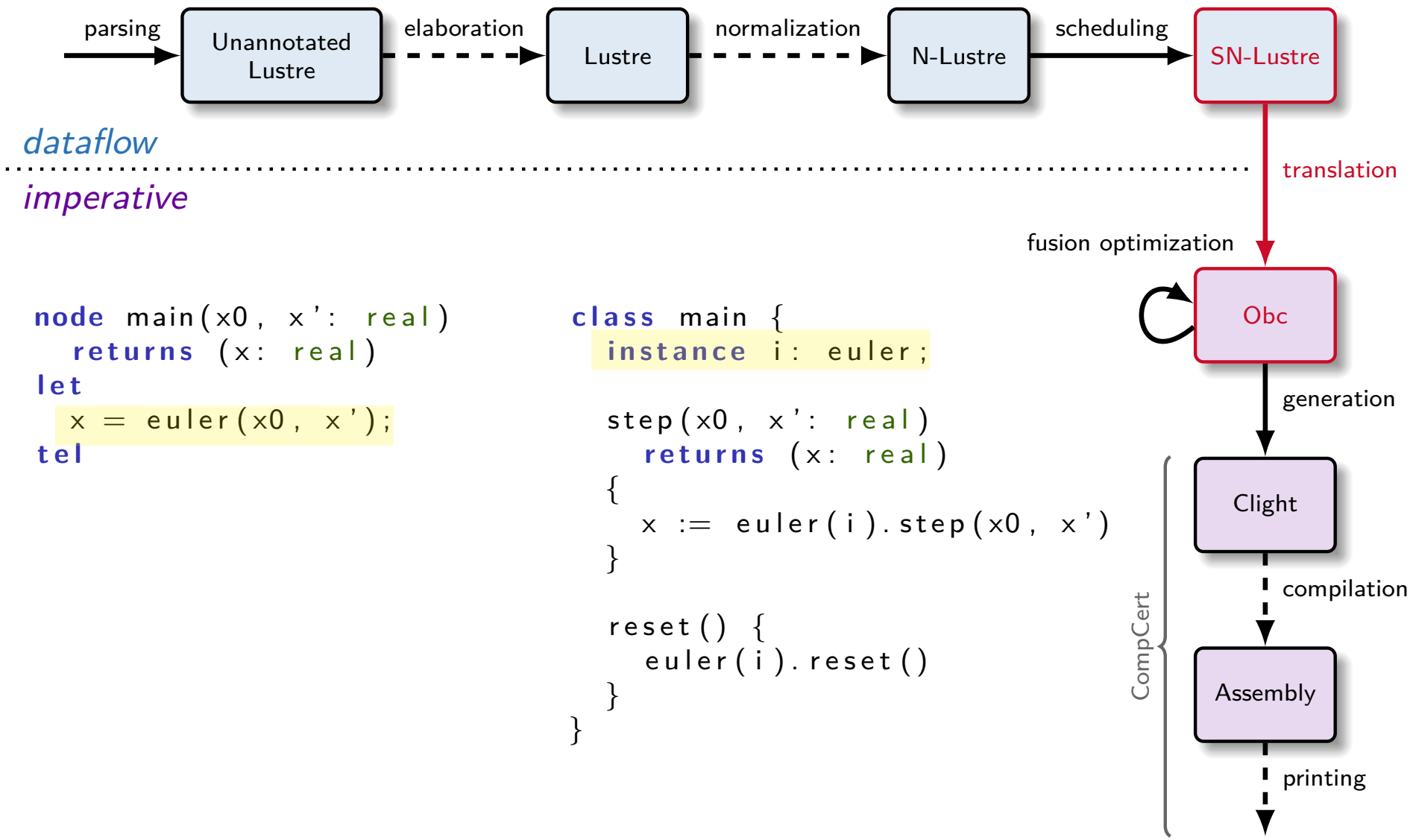
Vélus: a verified compiler




```
node main(x0, x': real)
  returns (x: real)
let
  x = euler(x0, x');
tel
```

```
class main {
  instance i: euler;

  step(x0, x': real)
    returns (x: real)
  {
    x := euler(i).step(x0, x')
  }

  reset() {
    euler(i).reset()
  }
}
```

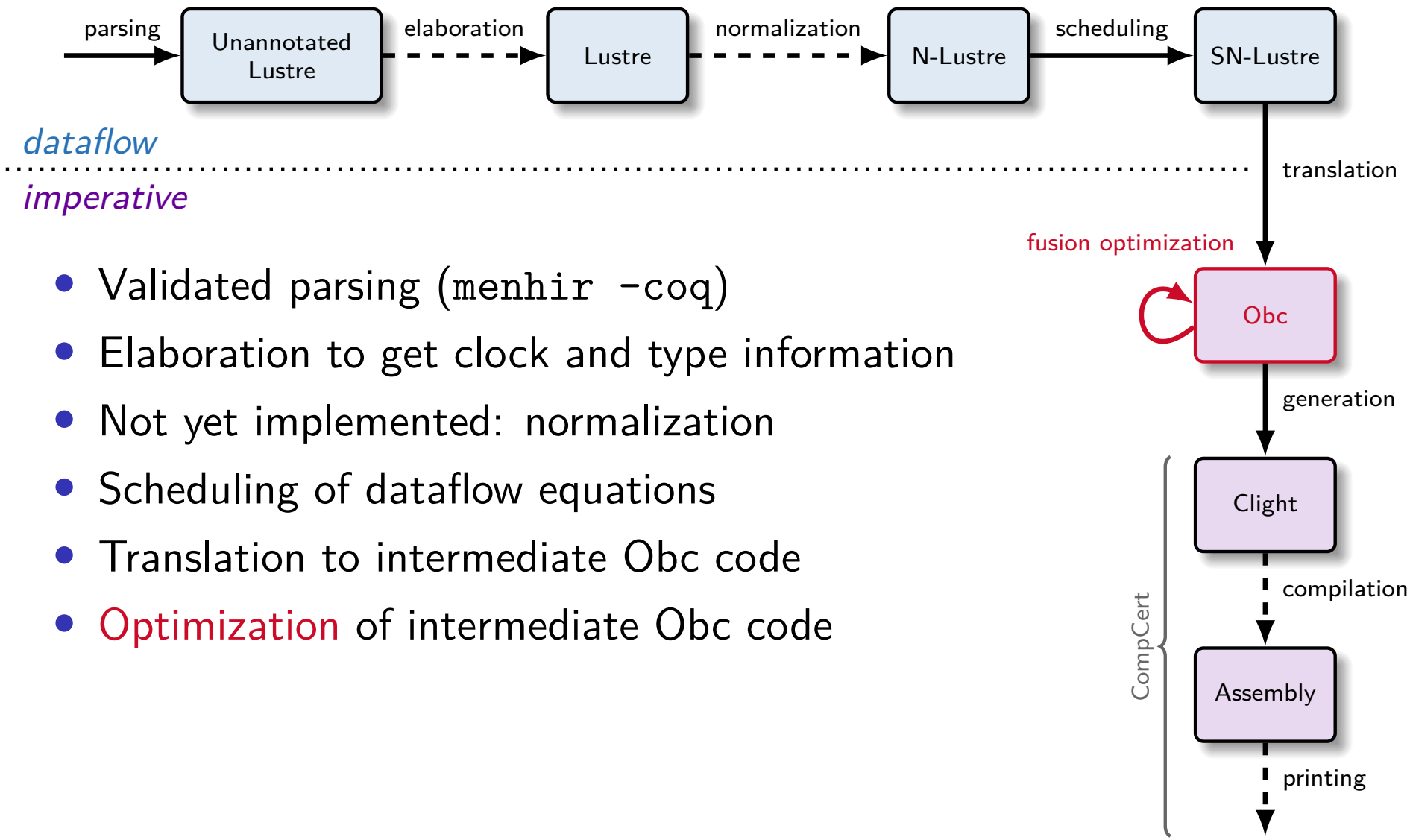
Vélus: a verified compiler




```
node main(x0, x': real)
  returns (x: real)
let
  x = euler(x0, x');
tel
```

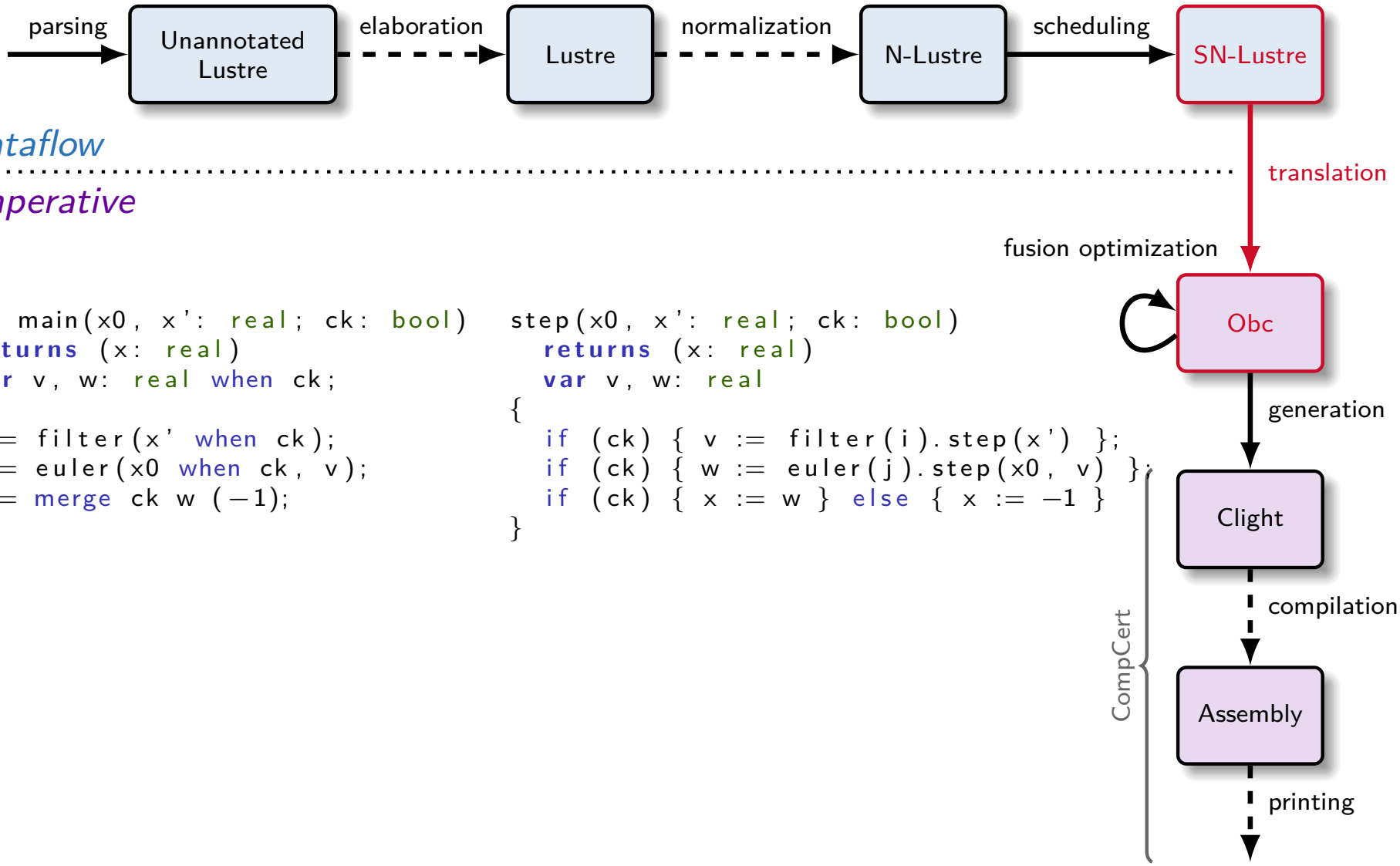
```
class main {
  instance i: euler;

  step(x0, x': real)
    returns (x: real)
  {
    x := euler(i).step(x0, x')
  }

  reset() {
    euler(i).reset()
  }
}
```

Vélus: a verified compiler



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

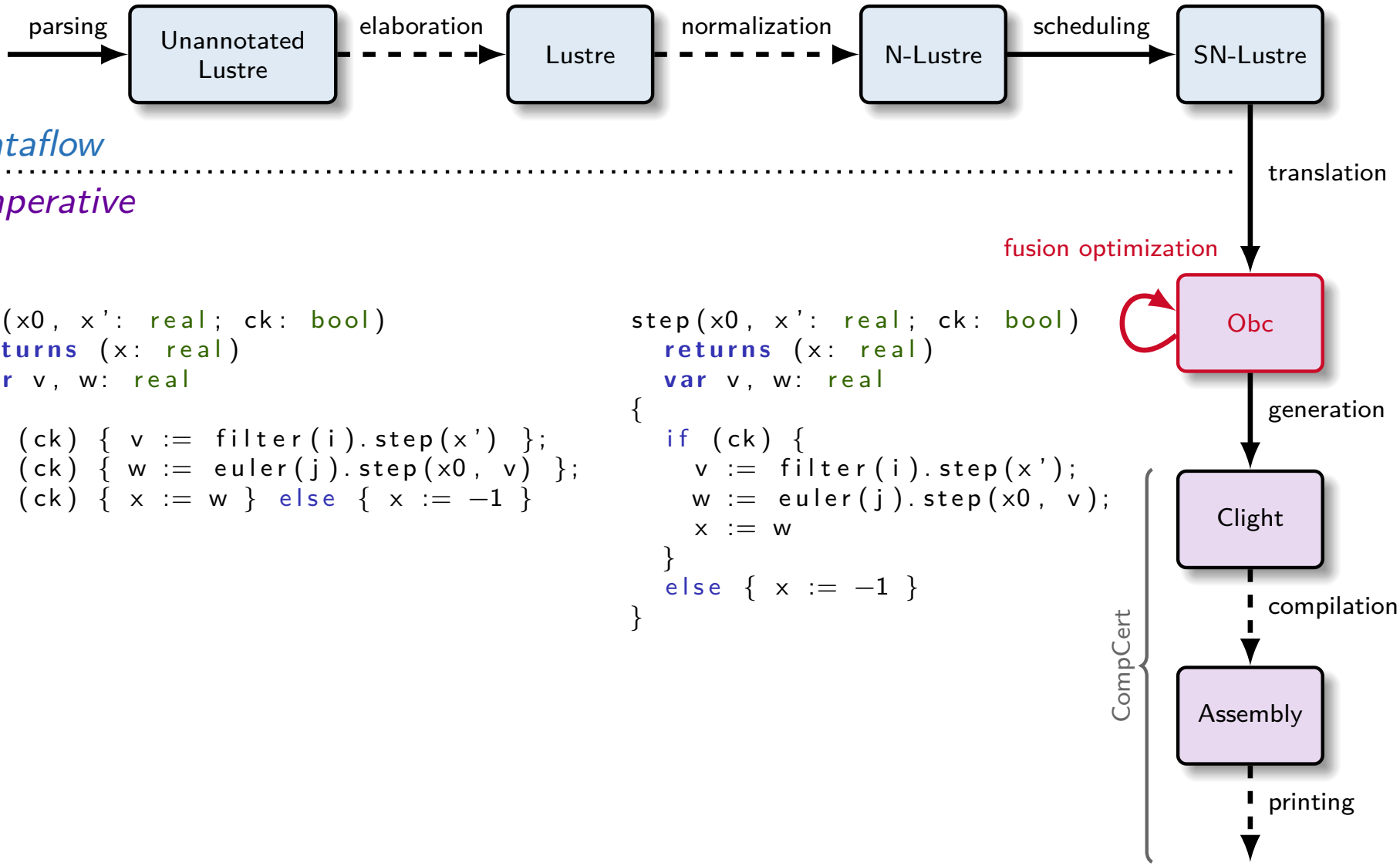
Vélus: a verified compiler




```
node main(x0, x': real; ck: bool)
  returns (x: real)
  var v, w: real when ck;
let
  v = filter(x' when ck);
  w = euler(x0 when ck, v);
  x = merge ck w (-1);
tel
```

```
step(x0, x': real; ck: bool)
  returns (x: real)
  var v, w: real
{
  if (ck) { v := filter(i).step(x') };
  if (ck) { w := euler(j).step(x0, v) };
  if (ck) { x := w } else { x := -1 }
}
```

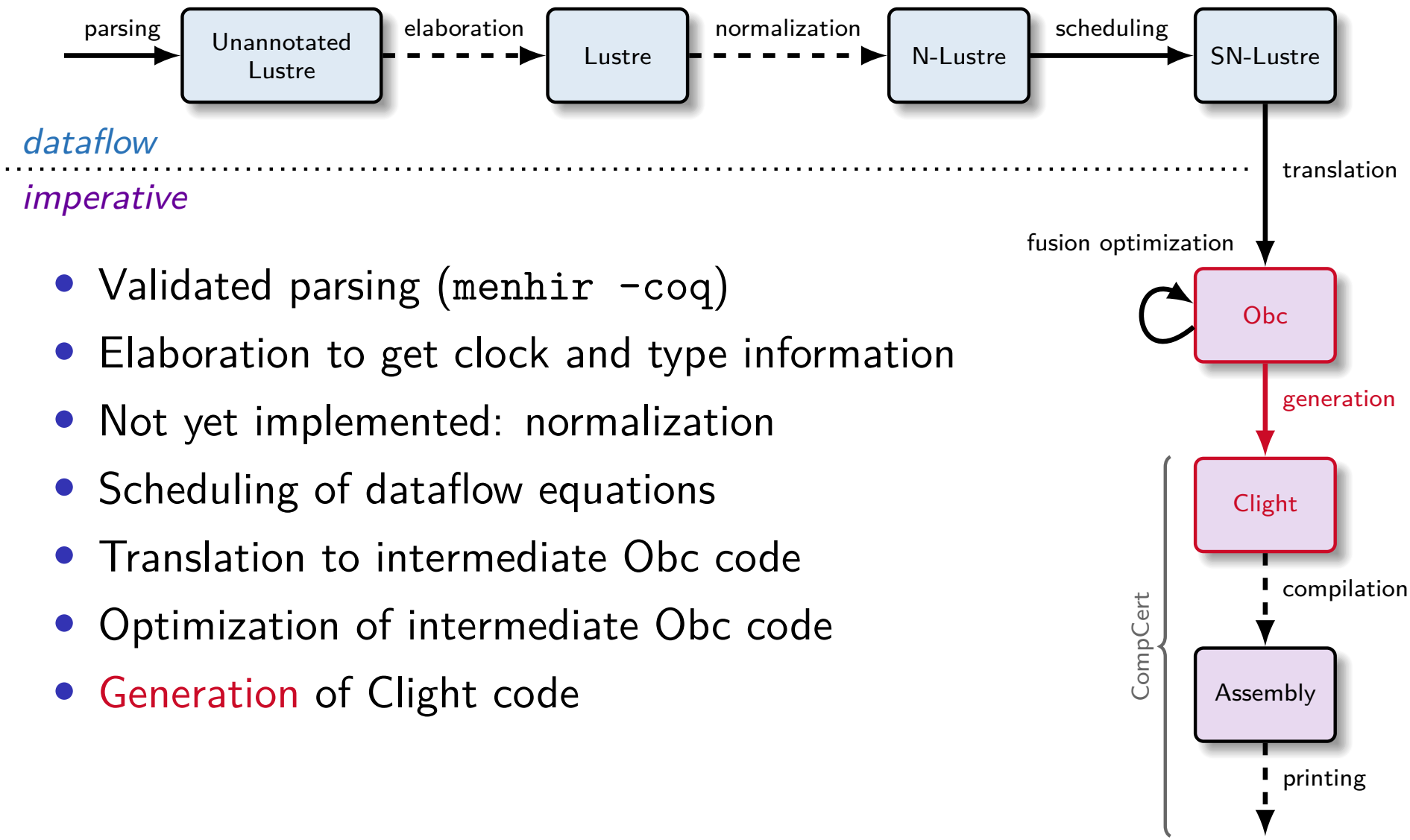
Vélus: a verified compiler




```
step(x0, x': real; ck: bool)
  returns (x: real)
  var v, w: real
{
  if (ck) { v := filter(i).step(x') };
  if (ck) { w := euler(j).step(x0, v) };
  if (ck) { x := w } else { x := -1 }
}
```

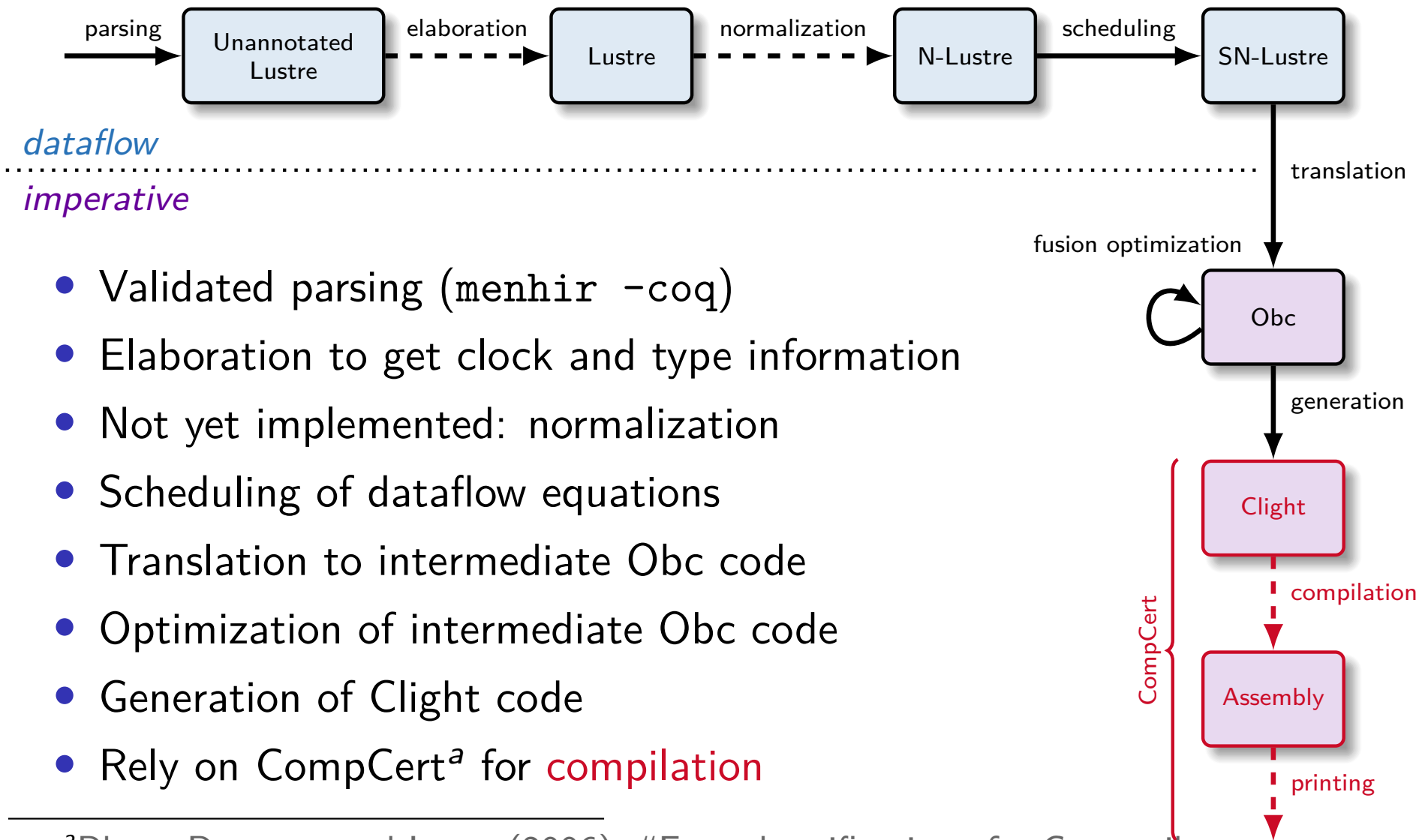
```
step(x0, x': real; ck: bool)
  returns (x: real)
  var v, w: real
{
  if (ck) {
    v := filter(i).step(x');
    w := euler(j).step(x0, v);
    x := w
  }
  else { x := -1 }
}
```

Vélus: a verified compiler



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

First issue: naive compilation

```
y = f(x) every r;
```

```
if (ck_r) {
  if (r) { f(i).reset() };
};
y := f(i).step(x)
```

First issue: naive compilation

```
y = f(x) every r;
```

```
if (ck_r) {
  if (r) { f(i).reset() };
};
y := f(i).step(x)
```

Problem with fusion optimization:

```
node main(x0, x': real; ck, r: bool)
  returns (x: real)
  var v, w: real when ck;
let
  v = filter(x' when ck);
  w = euler((x0, v) when ck) every r;
  x = merge ck w 0;
tel
```

First issue: naive compilation

```
y = f(x) every r;
```

```
if (ck_r) {
  if (r) { f(i).reset() };
};
y := f(i).step(x)
```

Problem with fusion optimization:

```
node main(x0, x': real; ck, r: bool)
  returns (x: real)
  var v, w: real when ck;
let
  v = filter(x' when ck);
  w = euler((x0, v) when ck) every r;
  x = merge ck w 0;
tel
```

```
step(x0, x': real; ck, r: bool)
  returns (x: real)
  var v, w : real
{
  if (ck) { v := filter(i).step(x') };
  if (r) { euler(j).reset() };
  if (ck) { w := euler(j).step(x0, v) };
  if (ck) { x := w } else { x := 0 }
}
```

First issue: naive compilation

```
y = f(x) every r;
```

```
if (ck_r) {
  if (r) { f(i).reset() };
};
y := f(i).step(x)
```

Problem with fusion optimization:

```
node main(x0, x': real; ck, r: bool)
  returns (x: real)
  var v, w: real when ck;
let
  v = filter(x' when ck);
  w = euler((x0, v) when ck) every r;
  x = merge ck w 0;
tel
```

```
step(x0, x': real; ck, r: bool)
  returns (x: real) var v, w : real
{
  if (r) { euler(j).reset() };
  if (ck) {
    v := filter(i).step(x');
    w := euler(j).step(x0, v);
    x := w
  } else { x := 0 }
}
```

Second issue: proof of correctness

N-Lustre

$\overrightarrow{V^\omega} \times \overrightarrow{V^\omega}$

Obc

$S \times \overrightarrow{V} \to S \times \overrightarrow{V}$

Second issue: proof of correctness

N-Lustre

$\overrightarrow{V^\omega} \times \overrightarrow{V^\omega}$

too weak for a direct
proof by induction

Obc

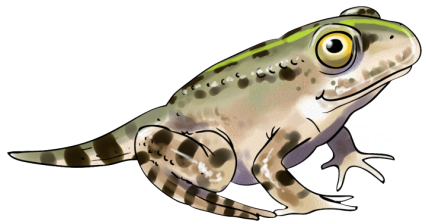
$S \times \overrightarrow{V} \to S \times \overrightarrow{V}$

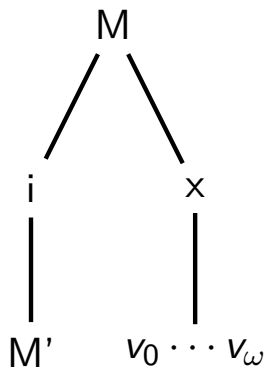
Second issue: proof of correctness

N-Lustre

$\overrightarrow{V^\omega} \times \overrightarrow{V^\omega}$

"easy": $\exists M$





Obc

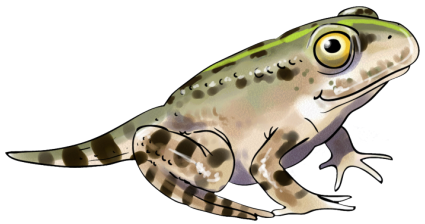
$S \times \overrightarrow{V} \to S \times \overrightarrow{V}$

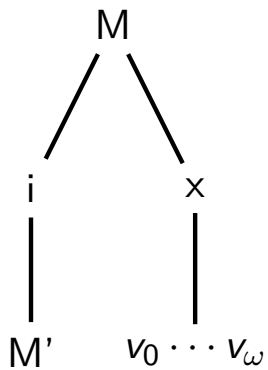
Second issue: proof of correctness

N-Lustre

$\overrightarrow{V^\omega} \times \overrightarrow{V^\omega}$

"easy": $\exists M$





"hard"

Obc

$S \times \overrightarrow{V} \to S \times \overrightarrow{V}$

Second issue: proof of correctness

What about the reset?

- It is possible to give a semantics in the intermediate semantics model as well

Second issue: proof of correctness

What about the reset?

- It is possible to give a semantics in the intermediate semantics model as well
- The “easy” proof can be done

Second issue: proof of correctness

What about the reset?

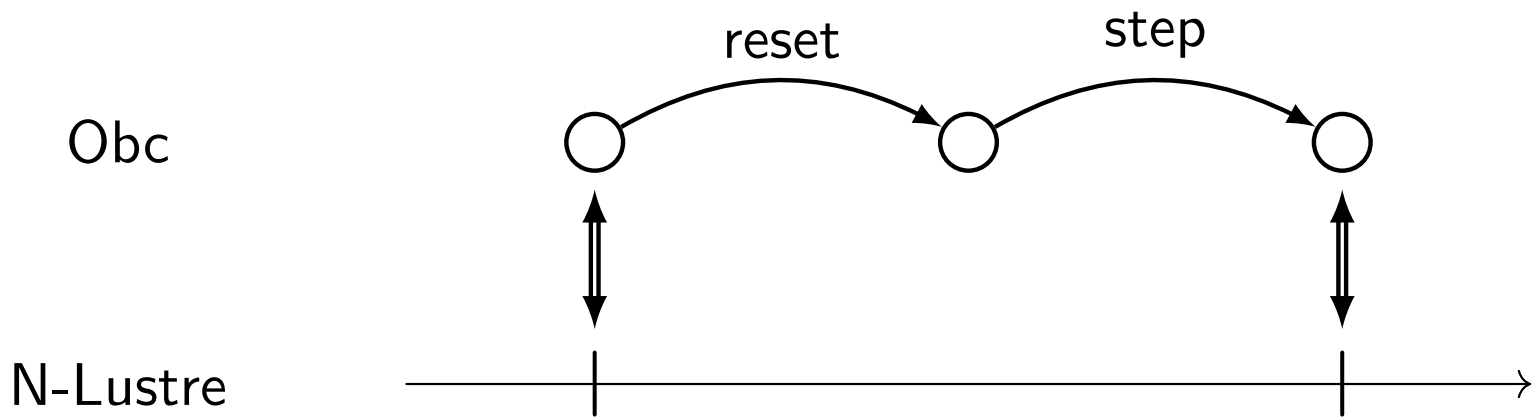
- It is possible to give a semantics in the intermediate semantics model as well
- The “easy” proof can be done
- The “hard” one cannot

Second issue: proof of correctness

The granularity of the intermediate model is not precise enough!

Second issue: proof of correctness

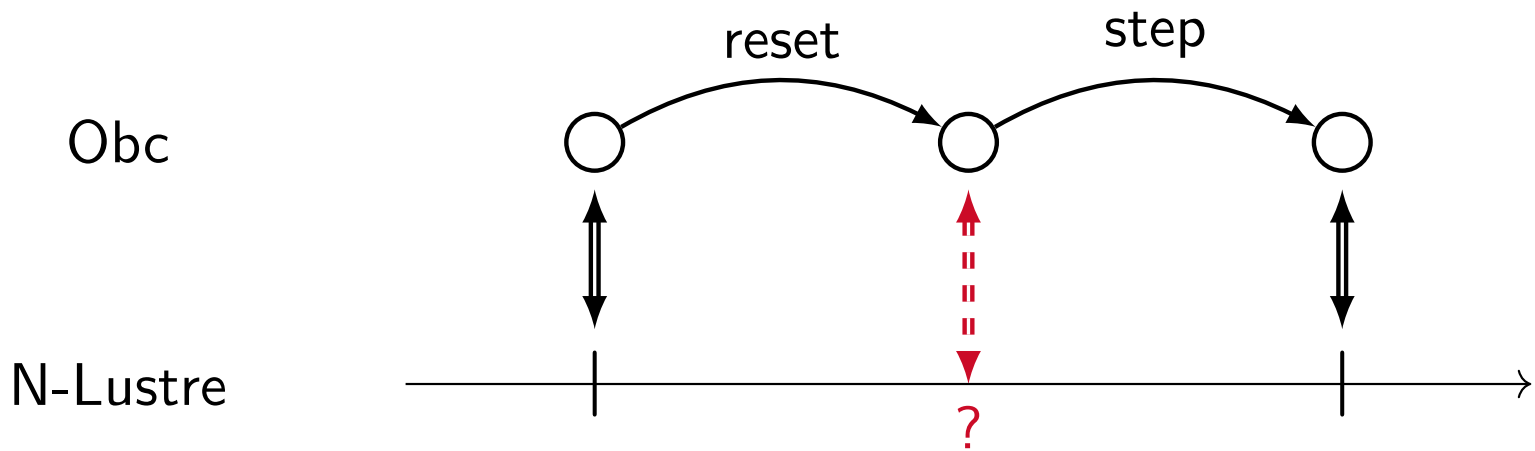
The granularity of the intermediate model is not precise enough!



Indeed, because of the reset, the memory can reach 2 different states within the same synchronous instant

Second issue: proof of correctness

The granularity of the intermediate model is not precise enough!



Indeed, because of the reset, the memory can reach 2 different states within the same synchronous instant

$\rightarrow$ Transient states

SyBloc

- Problems with the compilation scheme
- Problems with the semantics models

SyBloc

- Problems with the compilation scheme
- Problems with the semantics models

Propose a new intermediate language

- Declarative, like N-Lustre

SyBloc

- Problems with the compilation scheme
- Problems with the semantics models

Propose a new intermediate language

- Declarative, like N-Lustre
- Where the reset construct is treated as a separate “equation”

SyBloc

- Problems with the compilation scheme
- Problems with the semantics models

Propose a new intermediate language

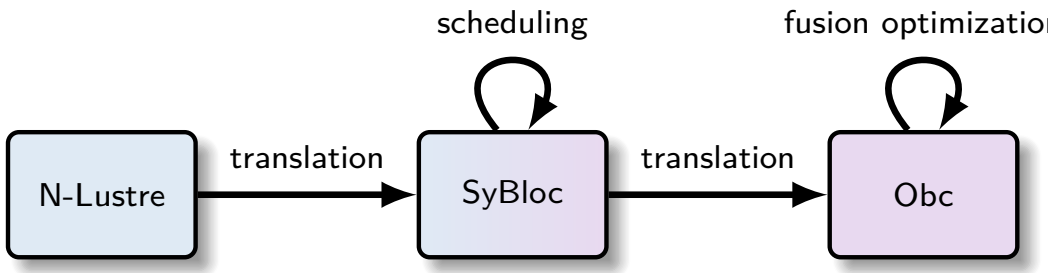
- Declarative, like N-Lustre
- Where the reset construct is treated as a separate “equation”
- Mimicking the intermediate memory model of N-Lustre

SyBloc

- Problems with the compilation scheme
- Problems with the semantics models

Propose a new intermediate language

- Declarative, like N-Lustre
- Where the reset construct is treated as a separate “equation”
- Mimicking the intermediate memory model of N-Lustre



N-Lustre

SyBloc

```
node euler(y0, y': real)
  returns (y: real)
  var h: real;
let
  y = y0 fby (y + y' * h);
  h = 2;
tel

node main(x0, x': real)
  returns (x: real)
  var r: bool;
let
  x = euler(x0, x') every r;
  r = (x' > 42);
tel
```

N-Lustre

```
node euler(y0, y': real)
  returns (y: real)
  var h: real;
let
  y = y0 fby (y + y' * h);
  h = 2;
tel

node main(x0, x': real)
  returns (x: real)
  var r: bool;
let
  x = euler(x0, x') every r;
  r = (x' > 42);
tel
```

SyBloc

```
block euler(y0, y': real)
  returns (y: real)
  var h: real;
let
  y = y0 fby (y + y' * h);
  h = 2;
tel

block main(x0, x': real)
  returns (x: real)
  var r: bool;
  instance i: euler;
let
  () = i.reset_on r;
  x = i.euler(x0, x');
  r = (x' > 42);
tel
```

N-Lustre

```
node euler(y0, y': real)
  returns (y: real)
  var h: real;
let
  y = y0 fby (y + y' * h);
  h = 2;
tel

node main(x0, x': real)
  returns (x: real)
  var r: bool;
let
  x = euler(x0, x') every r;
  r = (x' > 42);
tel
```

SyBloc

```
block euler(y0, y': real)
  returns (y: real)
  var h: real;
let
  y = y0 fby (y + y' * h);
  h = 2;
tel

block main(x0, x': real)
  returns (x: real)
  var r: bool;
  instance i: euler;
let
  () = i.reset_on r;
  x = i.euler(x0, x');
  r = (x' > 42);
tel
```

N-Lustre

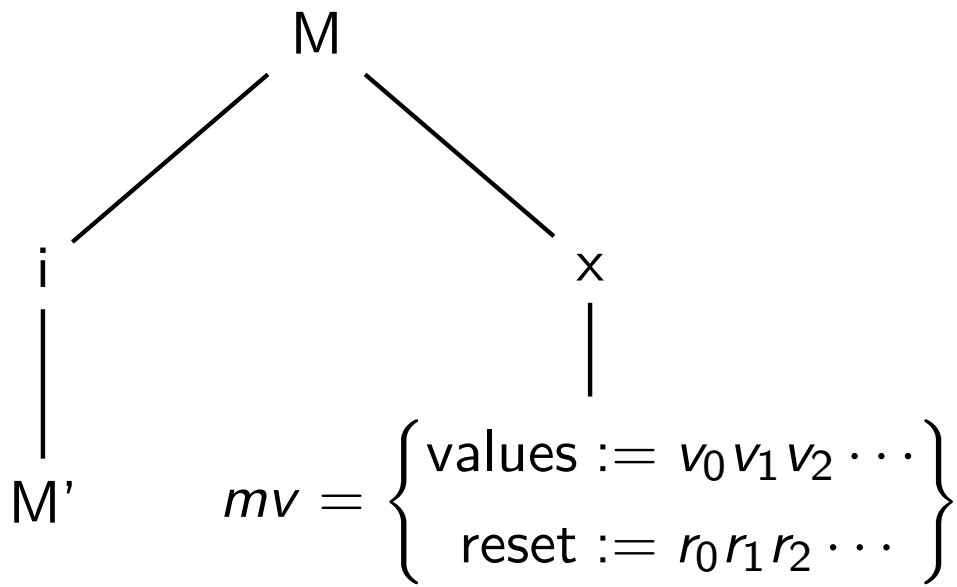
```
node euler(y0, y': real)
  returns (y: real)
  var h: real;
let
  y = y0 fby (y + y' * h);
  h = 2;
tel

node main(x0, x': real)
  returns (x: real)
  var r: bool;
let
  x = euler(x0, x') every r;
  r = (x' > 42);
tel
```

SyBloc

```
block euler(y0, y': real)
  returns (y: real)
  var h: real;
let
  y = y0 fby (y + y' * h);
  h = 2;
tel

block main(x0, x': real)
  returns (x: real)
  var r: bool;
  instance i: euler;
let
  () = i.reset_on r;
  x = i.euler(x0, x');
  r = (x' > 42);
tel
```

SyBloc formal semantics: memory

A tree with instances as nodes and the values of the fbys together with a boolean reset flag at the leaves :
{ values: *stream val*; reset: *stream bool* }



Notation:

instance $M[i] = M'$

register $M(x) = mv$

SyBloc formal semantics: fby

$$\frac{}{M \vdash_{\mathsf{eqn}} x = c_0 \text{ fby } e}$$

SyBloc formal semantics: fby

$$\frac{\vdash_{\mathsf{const}} c_0 \Downarrow v_0}{M \vdash_{\mathsf{eqn}} x = c_0 \text{ fby } e}$$

SyBloc formal semantics: fby

$$\frac{\vdash_{\mathsf{const}} c_0 \Downarrow v_0 \qquad \vdash_{\mathsf{exp}} e \Downarrow vs}{M \vdash_{\mathsf{eqn}} x = c_0 \text{ fby } e}$$

SyBloc formal semantics: fby

$$\frac{\vdash_{\mathsf{const}} c_0 \Downarrow v_0 \qquad \vdash_{\mathsf{exp}} e \Downarrow vs \qquad \vdash_{\mathsf{var}} x \Downarrow xs}{M \vdash_{\mathsf{eqn}} x = c_0 \text{ fby } e}$$

SyBloc formal semantics: fby

$$\frac{\vdash_{\mathsf{const}} c_0 \Downarrow v_0 \qquad \vdash_{\mathsf{exp}} e \Downarrow vs \qquad \vdash_{\mathsf{var}} x \Downarrow xs \qquad \mathsf{mfby}\ x\ v_0\ vs\ M\ xs}{M \vdash_{\mathsf{eqn}} x = c_0\ \mathsf{fby}\ e}$$

SyBloc formal semantics: fby

$$\frac{\vdash_{\text{const}} c_0 \Downarrow v_0 \qquad \vdash_{\text{exp}} e \Downarrow vs \qquad \vdash_{\text{var}} x \Downarrow xs \qquad \text{mfby } x\ v_0\ vs\ M\ xs}{M \vdash_{\text{eqn}} x = c_0 \text{ fby } e}$$

$$\frac{}{\text{mfby } x\ v_0\ vs\ M\ xs}$$

SyBloc formal semantics: fby

$$\frac{\vdash_{\mathsf{const}} c_0 \Downarrow v_0 \qquad \vdash_{\mathsf{exp}} e \Downarrow vs \qquad \vdash_{\mathsf{var}} x \Downarrow xs \qquad \mathsf{mfby}\ x\ v_0\ vs\ M\ xs}{M \vdash_{\mathsf{eqn}} x = c_0\ \mathsf{fby}\ e}$$

$$\frac{M(x) = mv}{\mathsf{mfby}\ x\ v_0\ vs\ M\ xs}$$

SyBloc formal semantics: fby

$$\frac{\vdash_{\text{const}} c_0 \Downarrow v_0 \qquad \vdash_{\text{exp}} e \Downarrow vs \qquad \vdash_{\text{var}} x \Downarrow xs \qquad \text{mfby } x\ v_0\ vs\ M\ xs}{M \vdash_{\text{eqn}} x = c_0 \text{ fby } e}$$

$$\frac{M(x) = mv \qquad mv.\text{values } 0 = v_0}{\text{mfby } x\ v_0\ vs\ M\ xs}$$

SyBloc formal semantics: fby

$$\frac{\vdash_{\text{const}} c_0 \Downarrow v_0 \qquad \vdash_{\text{exp}} e \Downarrow vs \qquad \vdash_{\text{var}} x \Downarrow xs \qquad \text{mfby } x\ v_0\ vs\ M\ xs}{M \vdash_{\text{eqn}} x = c_0\ \text{fby}\ e}$$

$$\frac{M(x) = mv \qquad mv.\text{values}\ 0 = v_0 \qquad \forall n, \text{fbyspec}\ n\ v_0\ (vs\ n)\ mv\ (xs\ n)}{\text{mfby}\ x\ v_0\ vs\ M\ xs}$$

SyBloc formal semantics: fby

$$\frac{\vdash_{\text{const}} c_0 \Downarrow v_0 \qquad \vdash_{\text{exp}} e \Downarrow vs \qquad \vdash_{\text{var}} x \Downarrow xs \qquad \text{mfby } x\ v_0\ vs\ M\ xs}{M \vdash_{\text{eqn}} x = c_0 \text{ fby } e}$$

$$\frac{M(x) = mv \qquad mv.\text{values } 0 = v_0 \qquad \forall n, \text{fbyspec } n\ v_0\ (vs\ n)\ mv\ (xs\ n)}{\text{mfby } x\ v_0\ vs\ M\ xs}$$

$$\frac{}{\text{fbyspec } n\ v_0\ v\ mv\ (mv.\text{values } n)}$$

SyBloc formal semantics: fby

$$\frac{\vdash_{\text{const}} c_0 \Downarrow v_0 \qquad \vdash_{\text{exp}} e \Downarrow vs \qquad \vdash_{\text{var}} x \Downarrow xs \qquad \text{mfby } x\ v_0\ vs\ M\ xs}{M \vdash_{\text{eqn}} x = c_0\ \text{fby}\ e}$$

$$\frac{M(x) = mv \qquad mv.\text{values}\ 0 = v_0 \qquad \forall n, \text{fbyspec}\ n\ v_0\ (vs\ n)\ mv\ (xs\ n)}{\text{mfby}\ x\ v_0\ vs\ M\ xs}$$

$$\frac{\text{next}\ n\ mv\ v_0\ v}{\text{fbyspec}\ n\ v_0\ v\ mv\ (mv.\text{values}\ n)}$$

$$\text{next}\ n\ mv\ v_0\ v := mv.\text{values}\ (n+1) = \begin{cases} v_0 & \text{if } mv.\text{reset}\ (n+1), \\ v & \text{otherwise.} \end{cases}$$

SyBloc formal semantics: fby

$$\frac{\vdash_{\text{const}} c_0 \Downarrow v_0 \qquad \vdash_{\text{exp}} e \Downarrow vs \qquad \vdash_{\text{var}} x \Downarrow xs \qquad \text{mfby } x\ v_0\ vs\ M\ xs}{M \vdash_{\text{eqn}} x = c_0 \text{ fby } e}$$

$$\frac{M(x) = mv \qquad mv.\text{values } 0 = v_0 \qquad \forall n, \text{fbyspec } n\ v_0\ (vs\ n)\ mv\ (xs\ n)}{\text{mfby } x\ v_0\ vs\ M\ xs}$$

$$\frac{\text{next } n\ mv\ v_0\ v}{\text{fbyspec } n\ v_0\ v\ mv\ (mv.\text{values } n)} \qquad \frac{}{\text{fbyspec } n\ v_0\ -\ mv\ -}$$

$$\text{next } n\ mv\ v_0\ v := mv.\text{values } (n+1) = \begin{cases} v_0 & \text{if } mv.\text{reset } (n+1), \\ v & \text{otherwise.} \end{cases}$$

SyBloc formal semantics: fby

$$\frac{\vdash_{\text{const}} c_0 \Downarrow v_0 \qquad \vdash_{\text{exp}} e \Downarrow vs \qquad \vdash_{\text{var}} x \Downarrow xs \qquad \text{mfby } x\ v_0\ vs\ M\ xs}{M \vdash_{\text{eqn}} x = c_0 \text{ fby } e}$$

$$\frac{M(x) = mv \qquad mv.\text{values } 0 = v_0 \qquad \forall n, \text{fbyspec } n\ v_0\ (vs\ n)\ mv\ (xs\ n)}{\text{mfby } x\ v_0\ vs\ M\ xs}$$

$$\frac{\text{next } n\ mv\ v_0\ v}{\text{fbyspec } n\ v_0\ v\ mv\ (mv.\text{values } n)} \qquad \frac{\text{next } n\ mv\ v_0\ (mv.\text{values } n)}{\text{fbyspec } n\ v_0 - mv -}$$

$$\text{next } n\ mv\ v_0\ v := mv.\text{values } (n+1) = \begin{cases} v_0 & \text{if } mv.\text{reset } (n+1), \\ v & \text{otherwise.} \end{cases}$$

SyBloc formal semantics: reset

$$\frac{}{M \vdash_{\mathsf{eqn}} () = i.\mathsf{reset\_on}\ r}$$

SyBloc formal semantics: reset

$$\frac{M[i] = M'}{M \vdash_{\mathsf{eqn}} () = i.\mathsf{reset\_on}\ r}$$

SyBloc formal semantics: reset

$$\frac{M[i] = M' \qquad \vdash_{\mathsf{var}} r \Downarrow rs}{M \vdash_{\mathsf{eqn}} () = i.\mathsf{reset\_on}\ r}$$

SyBloc formal semantics: reset

$$\frac{M[i] = M' \qquad \vdash_{\mathsf{var}} r \Downarrow rs \qquad \mathsf{reset\_memory}\ (\mathsf{boolof}\ rs)\ M'}{M \vdash_{\mathsf{eqn}} () = i.\mathsf{reset\_on}\ r}$$

SyBloc formal semantics: reset

$$\frac{M[i] = M' \qquad \vdash_{\mathsf{var}} r \Downarrow rs \qquad \mathsf{reset\_memory}\ (\mathsf{boolof}\ rs)\ M'}{M \vdash_{\mathsf{eqn}} () = i.\mathsf{reset\_on}\ r}$$

$$\frac{}{\mathsf{reset\_memory}\ rk\ M}$$

SyBloc formal semantics: reset

$$\frac{M[i] = M' \qquad \vdash_{\mathsf{var}} r \Downarrow rs \qquad \mathsf{reset\_memory}\ (\mathsf{boolof}\ rs)\ M'}{M \vdash_{\mathsf{eqn}} () = i.\mathsf{reset\_on}\ r}$$

$$\frac{\forall x\ mv, M(x) = mv \rightarrow \forall n, rk\ n = \mathsf{true} \rightarrow mv.\mathsf{reset}\ n = true}{\mathsf{reset\_memory}\ rk\ M}$$

SyBloc formal semantics: reset

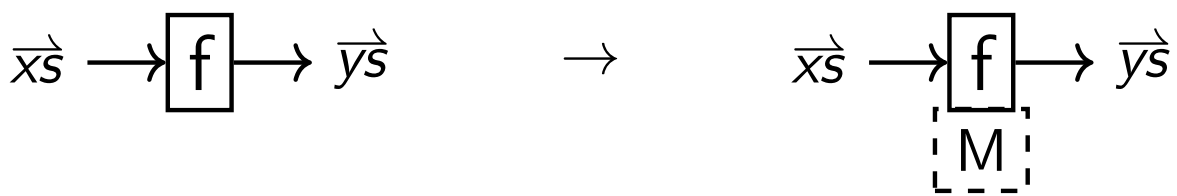
$$\frac{M[i] = M' \qquad \vdash_{\mathsf{var}} r \Downarrow rs \qquad \mathsf{reset\_memory}\ (\mathsf{boolof}\ rs)\ M'}{M \vdash_{\mathsf{eqn}} () = i.\mathsf{reset\_on}\ r}$$

$$\frac{\begin{array}{c}\forall i\ M', M[i] = M' \rightarrow \mathsf{reset\_memory}\ rk\ M' \\ \forall x\ mv, M(x) = mv \rightarrow \forall n, rk\ n = \mathsf{true} \rightarrow mv.\mathsf{reset}\ n = true\end{array}}{\mathsf{reset\_memory}\ rk\ M}$$

From N-Lustre to SyBloc

- Overall ideas of the "easy" proof + complications
- main theorem:

$$\begin{aligned}&\text{sem\_node } G\ f\ \overrightarrow{xs}\ \overrightarrow{ys} \rightarrow\\ &\exists M, \text{sem\_block } (\text{translate } G)\ f\ \overrightarrow{xs}\ M\ \overrightarrow{ys}\end{aligned}$$



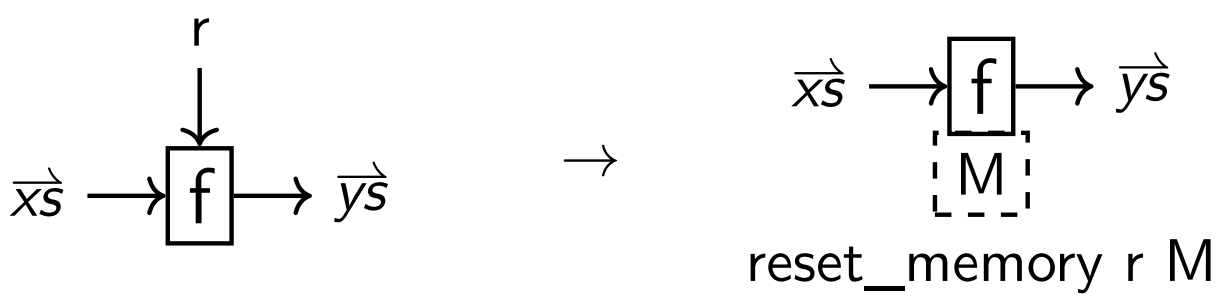
An issue within the mutual induction

Correctness for reset:

$$\left(\begin{array}{l} \quad \forall f\ \overrightarrow{xs}\ \overrightarrow{ys}, \\ IH: \quad \text{sem\_node}\ G\ f\ \overrightarrow{xs}\ \overrightarrow{ys} \rightarrow \\ \qquad \exists M, \text{sem\_block}\ (\text{translate}\ G)\ f\ \overrightarrow{xs}\ M\ \overrightarrow{ys} \end{array}\right) \rightarrow$$

$$\text{sem\_reset}\ G\ f\ r\ \overrightarrow{xs}\ \overrightarrow{ys} \rightarrow$$

$$\exists M, \begin{cases} \text{sem\_block}\ (\text{translate}\ G)\ f\ \overrightarrow{xs}\ M\ \overrightarrow{ys} \\ \text{reset\_memory}\ r\ M \end{cases}$$



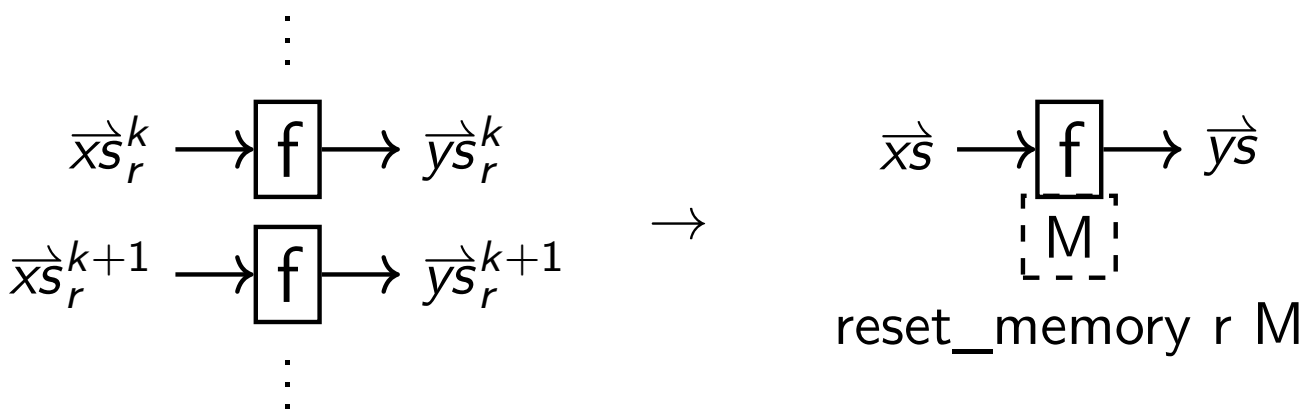
An issue within the mutual induction

Inversion on sem_reset:

$$\left(\begin{array}{l} \quad \forall f\ \overrightarrow{xs}\ \overrightarrow{ys}, \\ IH: \quad \text{sem\_node}\ G\ f\ \overrightarrow{xs}\ \overrightarrow{ys} \rightarrow \\ \quad\quad \exists M, \text{sem\_block}\ (\text{translate}\ G)\ f\ \overrightarrow{xs}\ M\ \overrightarrow{ys} \end{array}\right) \rightarrow$$

$$(\forall k, \text{sem\_node}\ G\ f\ (\text{mask}\ k\ r\ \overrightarrow{xs})\ (\text{mask}\ k\ r\ \overrightarrow{ys})) \rightarrow$$

$$\exists M, \begin{cases} \text{sem\_block}\ (\text{translate}\ G)\ f\ \overrightarrow{xs}\ M\ \overrightarrow{ys} \\ \text{reset\_memory}\ r\ M \end{cases}$$

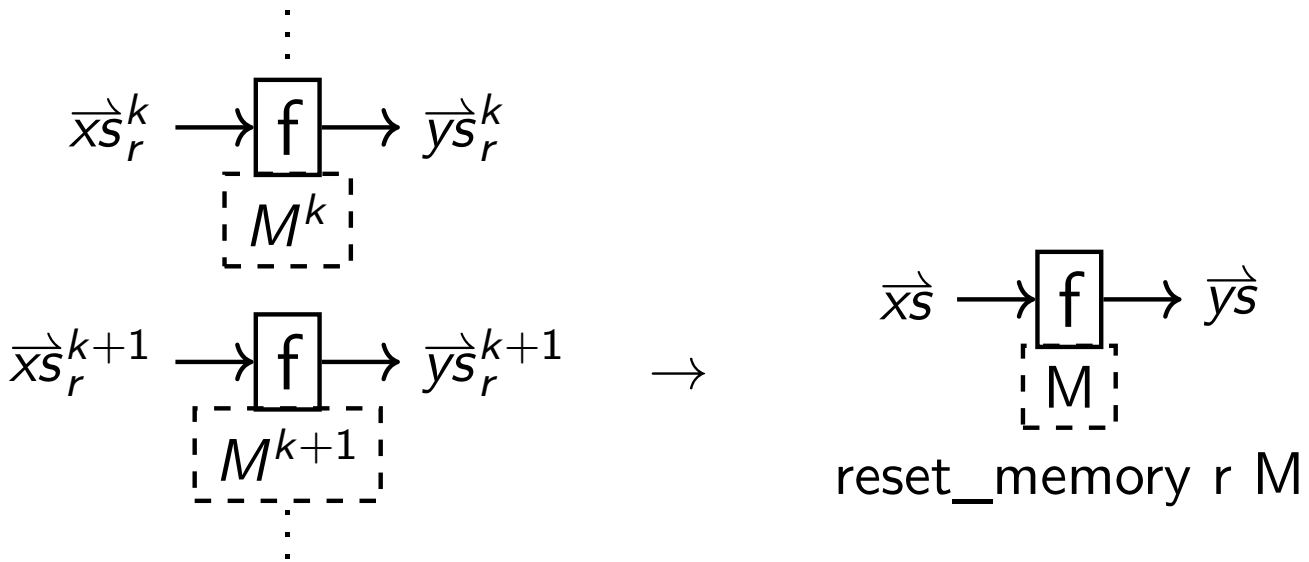


An issue within the mutual induction

Applying the mutual induction hypothesis:

$$\left(\forall k, \exists M^k, \text{sem\_block (translate } G) \ f \ (\text{mask } k \ r \ \overrightarrow{xs}) \ M^k \ (\text{mask } k \ r \ \overrightarrow{ys})\right) \rightarrow$$

$$\exists M, \begin{cases} \text{sem\_block (translate } G) \ f \ \overrightarrow{xs} \ M \ \overrightarrow{ys} \\ \text{reset\_memory } r \ M \end{cases}$$



An issue within the mutual induction

$$\left(\forall k, \exists M^k, \text{sem\_block (translate } G) \ f \ (\text{mask } k \ r \ \overrightarrow{xs}) \ M^k \ (\text{mask } k \ r \ \overrightarrow{ys})\right) \rightarrow$$

$$\exists M, \begin{cases} \text{sem\_block (translate } G) \ f \ \overrightarrow{xs} \ M \ \overrightarrow{ys} \\ \text{reset\_memory } r \ M \end{cases}$$

The idea

Reconstruct a memory from each memories given by each "slice" k

Axiom of choice?

- Our hypothesis:

$$\forall k, \exists M^k, \mathsf{sem\_block}\ (\mathsf{translate}\ G)\ f\ (\mathsf{mask}\ k\ r\ \overrightarrow{xs})\ M^k\ (\mathsf{mask}\ k\ r\ \overrightarrow{ys})$$

Axiom of choice?

- Our hypothesis:

$$\forall k, \exists M^k, \text{sem\_block}\ (\text{translate}\ G)\ f\ (\text{mask}\ k\ r\ \overrightarrow{xs})\ M^k\ (\text{mask}\ k\ r\ \overrightarrow{ys})$$

- The functional axiom of choice in Coq (standard library):

```
Axiom functional_choice :
  ∀ A B (R: A → B → Prop),
    (∀ x, ∃ y, R x y) →
    ∃ f, ∀ x, R x (f x).
```

Axiom of choice?

- Our hypothesis:

$$\forall k, \exists M^k, \text{sem\_block}\ (\text{translate}\ G)\ f\ (\text{mask}\ k\ r\ \overrightarrow{xs})\ M^k\ (\text{mask}\ k\ r\ \overrightarrow{ys})$$

- The functional axiom of choice in Coq (standard library):

```
Axiom functional_choice :
  ∀ A B (R: A → B → Prop),
    (∀ x, ∃ y, R x y) →
    ∃ f, ∀ x, R x (f x).
```

- Result:

$$\exists F_M, \forall k, \text{sem\_block}\ (\text{translate}\ G)\ f\ (\text{mask}\ k\ r\ \overrightarrow{xs})\ (F_M\ k)\ (\text{mask}\ k\ r\ \overrightarrow{ys})$$

Axiom of choice?

- Our hypothesis:

$$\forall k, \exists M^k, \text{sem\_block}\ (\text{translate}\ G)\ f\ (\text{mask}\ k\ r\ \overrightarrow{xs})\ M^k\ (\text{mask}\ k\ r\ \overrightarrow{ys})$$

- The functional axiom of choice in Coq (standard library):

```
Axiom functional_choice :
  ∀ A B (R: A → B → Prop),
    (∀ x, ∃ y, R x y) →
    ∃ f, ∀ x, R x (f x).
```

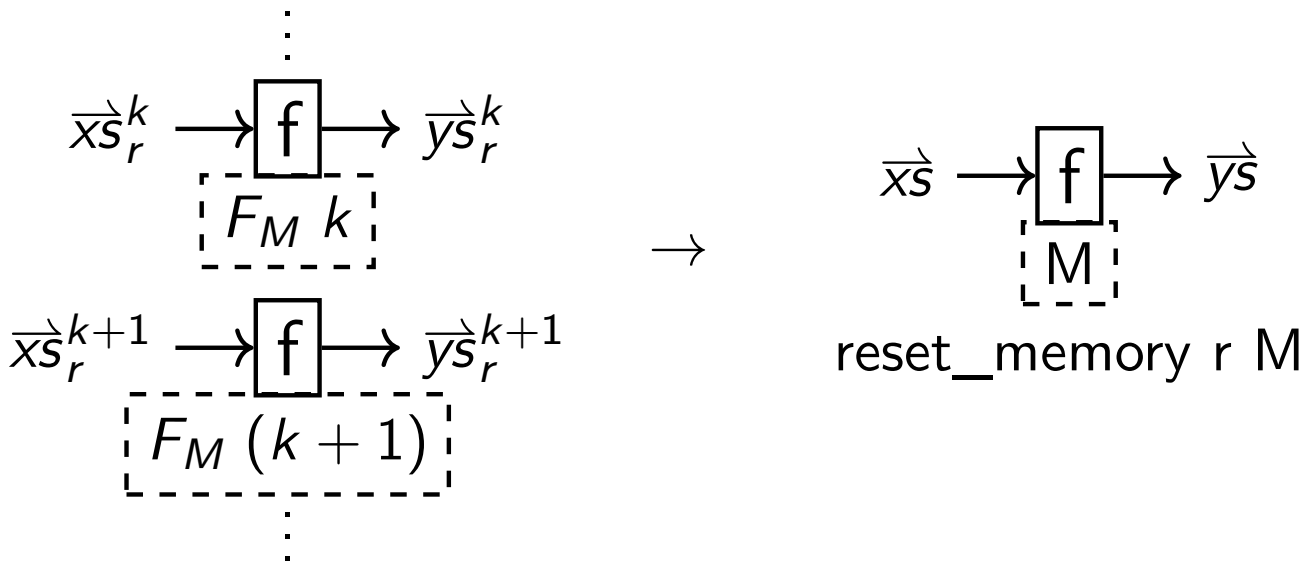
- Result:

$$\exists F_M, \forall k, \text{sem\_block}\ (\text{translate}\ G)\ f\ (\text{mask}\ k\ r\ \overrightarrow{xs})\ (F_M\ k)\ (\text{mask}\ k\ r\ \overrightarrow{ys})$$

Reconstructing the memory

$$(\forall k, \text{sem\_block } (\text{translate } G)\ f\ (\text{mask } k\ r\ \overrightarrow{xs})\ (F_M\ k)\ (\text{mask } k\ r\ \overrightarrow{ys})) \rightarrow$$

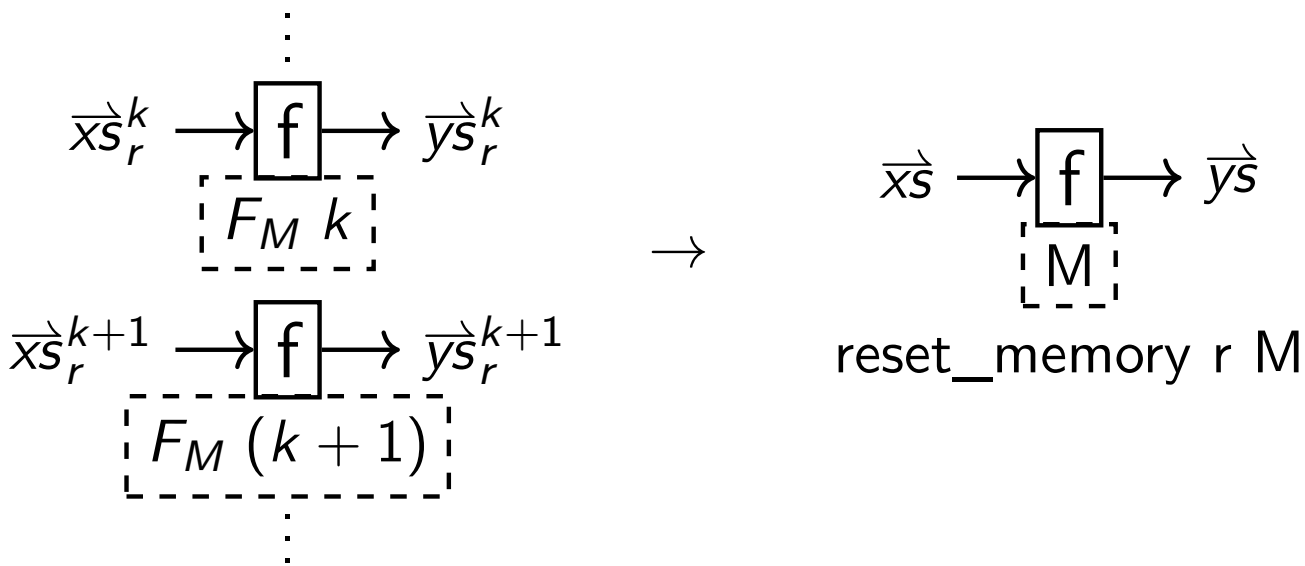
$$\exists M, \begin{cases} \text{sem\_block } (\text{translate } G)\ f\ \overrightarrow{xs}\ M\ \overrightarrow{ys} \\ \text{reset\_memory } r\ M \end{cases}$$



Reconstructing the memory

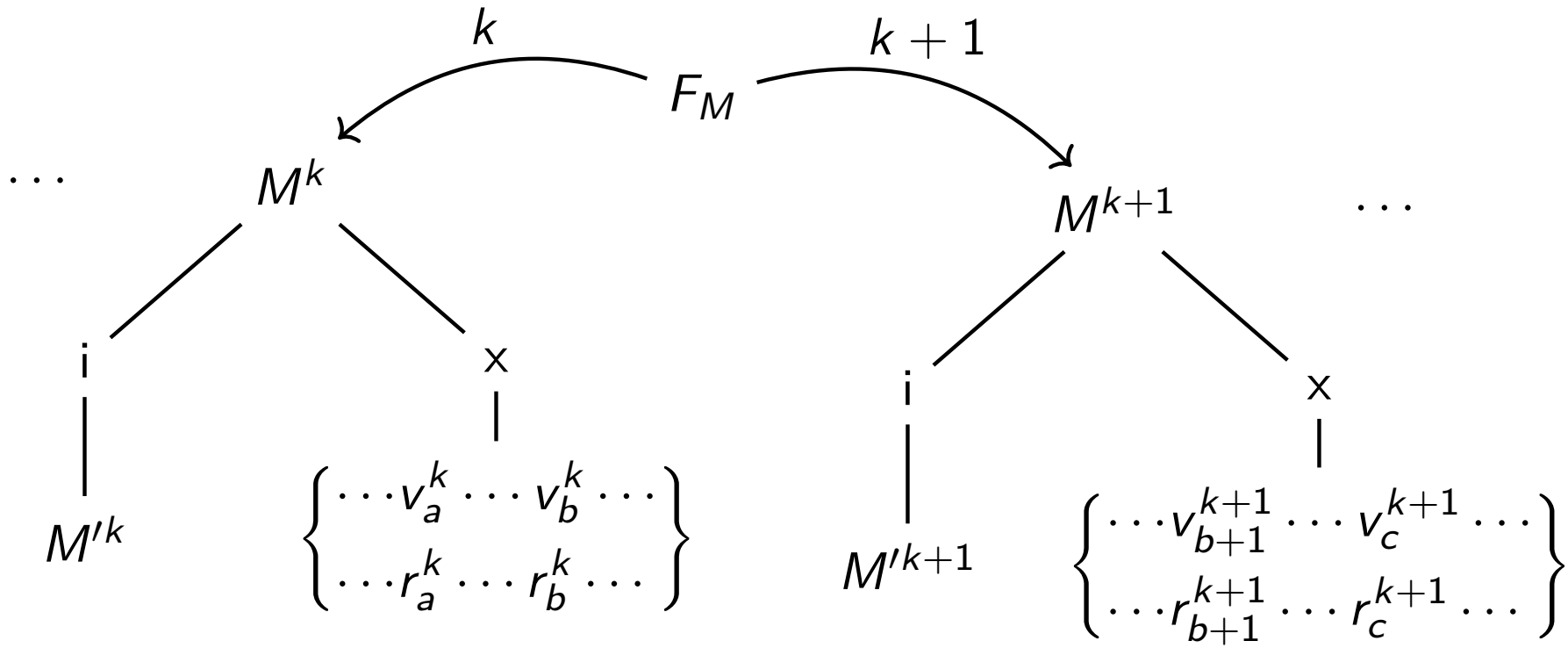
$$(\forall k, \text{sem\_block } (\text{translate } G)\ f\ (\text{mask } k\ r\ \overrightarrow{xs})\ (F_M\ k)\ (\text{mask } k\ r\ \overrightarrow{ys})) \rightarrow$$

$$\exists M, \begin{cases} \text{sem\_block } (\text{translate } G)\ f\ \overrightarrow{xs}\ M\ \overrightarrow{ys} \\ \text{reset\_memory } r\ M \end{cases}$$

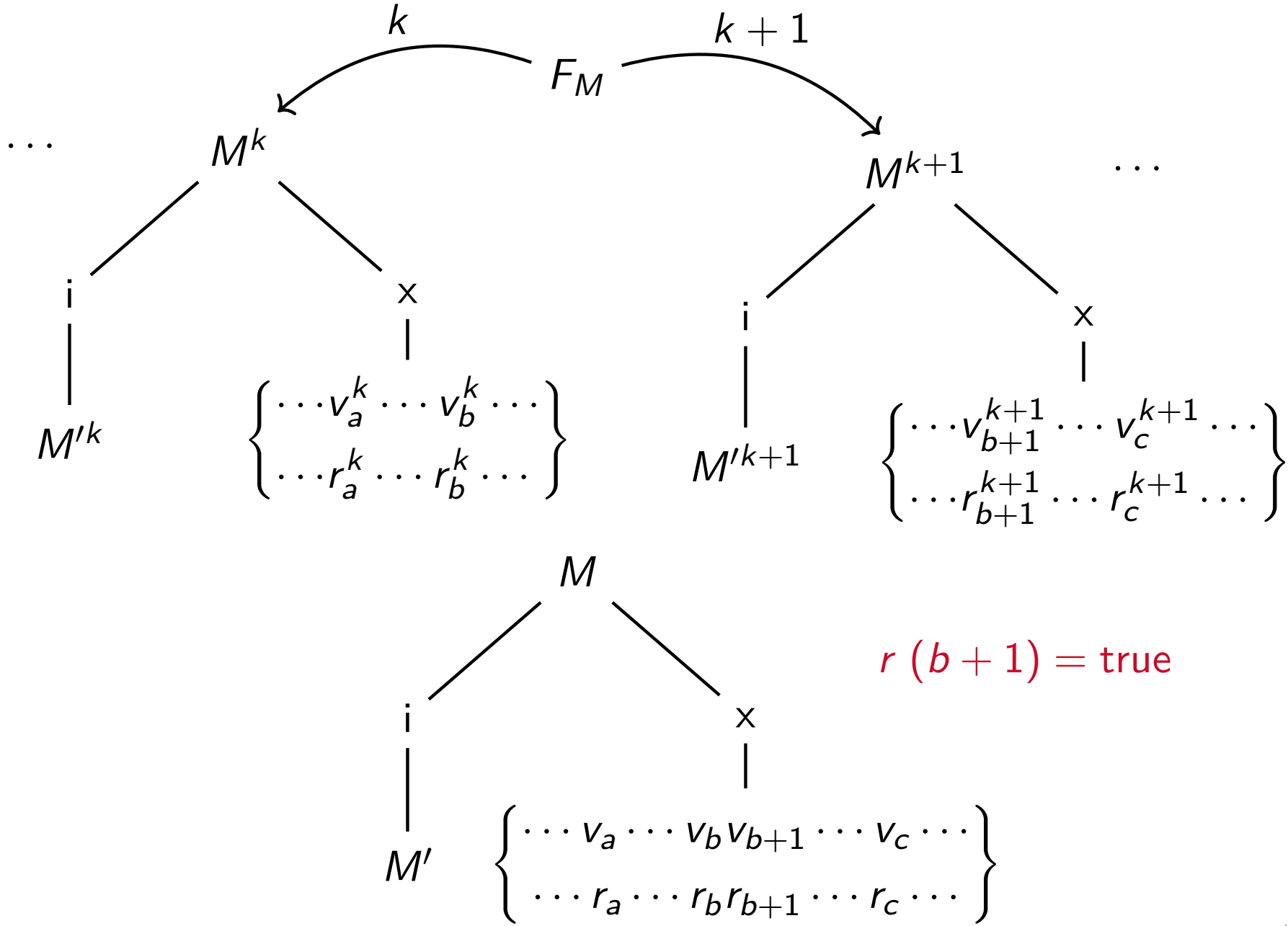


We can use F_M to build an M that verify the goal: at each instant n the content of M will be the same as $F_M\ k$ where k is the number of reset seen so far at n.

Reconstructing the memory

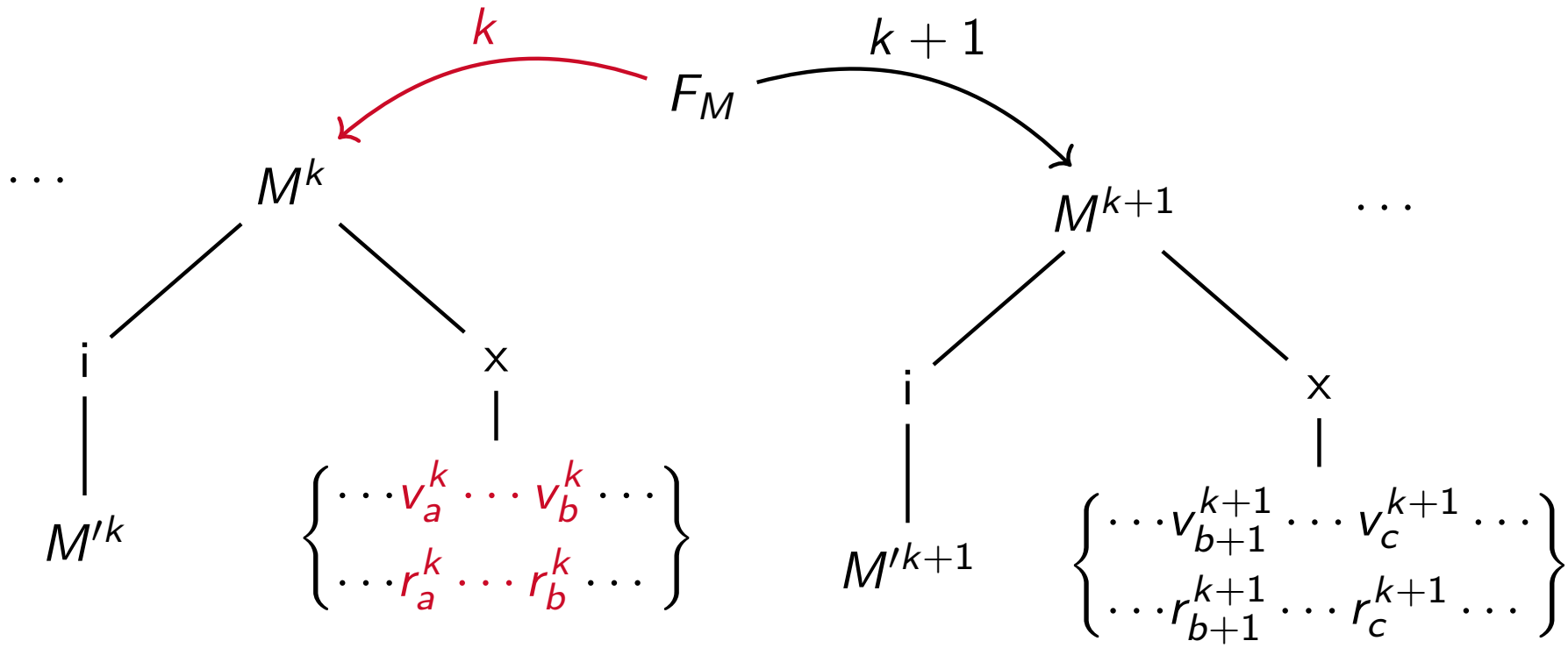


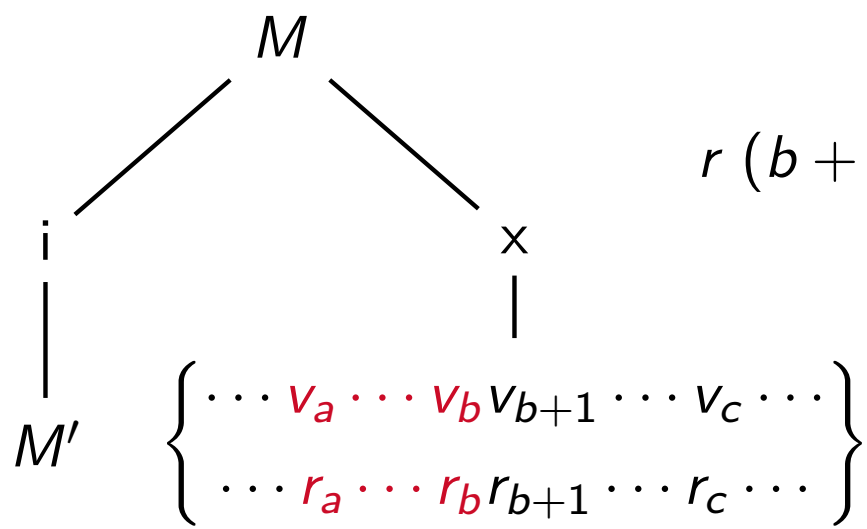
Reconstructing the memory



$r\,(b+1) = \text{true}$

Reconstructing the memory

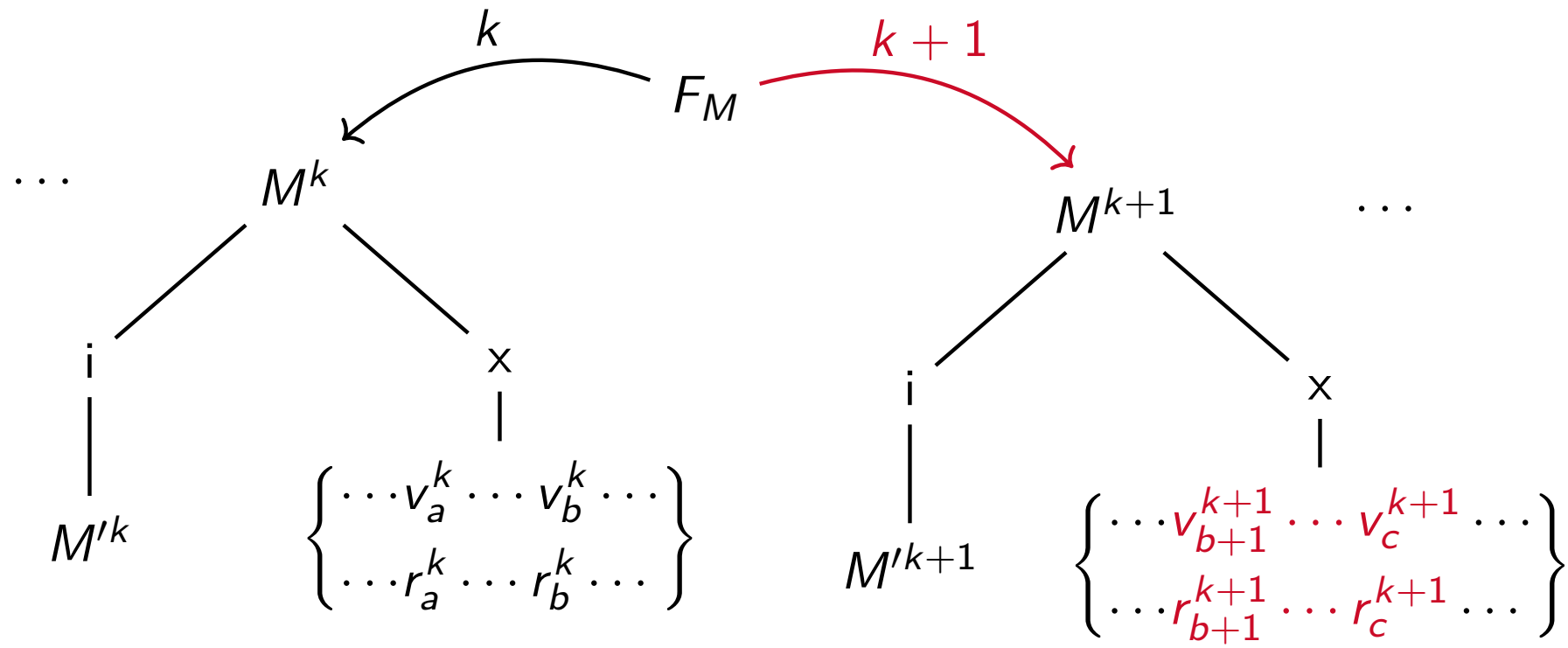


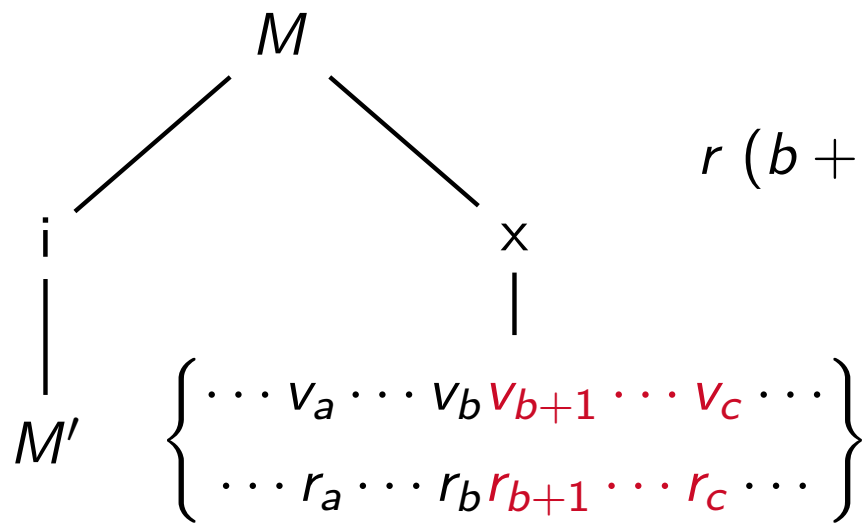


$r\ (b+1) = \text{true}$

$\forall n \in [a, b]$,
count $r\ n = k$
$r_n = r_n^k || r\ n$

Reconstructing the memory





$r\ (b+1) = \text{true}$

$\forall n \in]b, c]$,
count $r\ n = k+1$
$r_n = r_n^{k+1} || r\ n$

Last issue

reset correctness

$$\text{sem\_reset}\ G\ f\ r\ \overrightarrow{xs}\ \overrightarrow{ys} \rightarrow$$

$$\exists M, \begin{cases} \text{sem\_block}\ (\text{translate}\ G)\ f\ \overrightarrow{xs}\ M\ \overrightarrow{ys} \\ \text{reset\_memory}\ r\ M \end{cases}$$

Last issue

reset correctness

$$\text{sem\_reset}\ G\ f\ r\ \overrightarrow{xs}\ \overrightarrow{ys} \rightarrow$$
$$\exists M, \begin{cases} \text{sem\_block}\ (\text{translate}\ G)\ f\ \overrightarrow{xs}\ M\ \overrightarrow{ys} \\ \text{reset\_memory}\ r\ M \end{cases}$$

reset_memory

$$\frac{\forall i\ M', M[i] = M' \rightarrow \text{reset\_memory}\ rk\ M' \qquad \forall x\ mv, M(x) = mv \rightarrow \forall n, rk\ n = \text{true} \rightarrow mv.\text{reset}\ n = true}{\text{reset\_memory}\ rk\ M}$$

Last issue

reset correctness

$$\text{sem\_reset}\ G\ f\ r\ \overrightarrow{xs}\ \overrightarrow{ys} \rightarrow$$

$$\exists M, \begin{cases} \text{sem\_block}\ (\text{translate}\ G)\ f\ \overrightarrow{xs}\ M\ \overrightarrow{ys} \\ \text{reset\_memory}\ r\ M \end{cases}$$

reset_memory

$$\frac{\forall i\ M', M[i] = M' \rightarrow \text{reset\_memory}\ rk\ M' \qquad \forall x\ mv, M(x) = mv \rightarrow \forall n, rk\ n = \text{true} \rightarrow mv.\text{reset}\ n = true}{\text{reset\_memory}\ rk\ M}$$

Weak specification on reset flags: they could be true without an actual reset asking

Conclusion

Summary

- A verified compiler for Lustre
- Simple semantics for modular reset
- An intermediate “hybrid” language between dataflow and imperative worlds

Conclusion

Summary

- A verified compiler for Lustre
- Simple semantics for modular reset
- An intermediate "hybrid" language between dataflow and imperative worlds

Future Work

- Solving the issues
  - Is the semantics of SyBloc too weak?
  - Another model, more intricate?
  - Is the axiom of choice really mandatory?
  - Another way of formalizing the reset in N-Lustre?
- Complete the proof to Obc
- State machines

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:
```

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.
```

N-Lustre: abstract syntax

$le :=$ **expression**

k	(constant)
x	(variable)
le `when` x	(when)
$\diamond\ e$	(unary operator)
$e \oplus e$	(binary operator)

$ce :=$ **control expression**

`merge` x ce ce	(merge)
`if` x `then` ce `else` ce	(if)
le	(expression)

$eq :=$ **equation**

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

$n :=$ **node**

`node` $x(\overrightarrow{x^{ty::c}})$ `returns` $(\overrightarrow{x^{ty::c}})$
  [var $\overrightarrow{x^{ty::c}}$]
`let`
  $\overrightarrow{eq;}$
`tel`

Obc: Abstract Syntax

$e :=$ **expression**

- x (local variable)
- $\mathsf{state}(x)$ (state variable)
- k (constant)
- $\diamond\ e$ (unary operator)
- $e \oplus e$ (binary operator)

$s :=$ **statement**

- $x := e$ (update)
- $\mathsf{state}(x) := e$ (state update)
- $\mathsf{if}\ e\ \mathsf{then}\ s\ \mathsf{else}\ s$ (conditional)
- $\vec{x} := k(i).f(\vec{e})$ (method call)
- $s;\ s$ (composition)
- skip (do nothing)

$cls :=$ **declaration**

- $\mathsf{class}\ k\ \{$
  $\quad \mathsf{memory}\ \overrightarrow{x^{ty}}$
  $\quad \mathsf{instance}\ \overrightarrow{i^{k}}$
  $\quad f(\overrightarrow{x^{ty}})\ \mathsf{returns}\ (\overrightarrow{x^{ty}})\ [\mathsf{var}\ \overrightarrow{x^{ty}}]\ \{\ s\ \}$
  $\}$ (class)

Separation logic in CompCert

predicate

$$massert \triangleq \left\{ \begin{array}{l} \mathsf{pred} : memory \to \mathbb{P} \\ \mathsf{foot} : block \to int \to \mathbb{P} \\ \mathsf{invar} : \forall m\, m', \mathsf{pred}\ m \to \\ \qquad\qquad \mathsf{unchanged\_on}\ \mathsf{foot}\ m\ m' \to \\ \qquad\qquad \mathsf{pred}\ m' \end{array} \right\}$$

notation: $m \vDash P \triangleq P.\mathsf{pred}\ m$

Separation logic in CompCert

predicate

$$massert \triangleq \left\{ \begin{array}{l} \text{pred} : memory \to \mathbb{P} \\ \text{foot} : block \to int \to \mathbb{P} \\ \text{invar} : \forall m\, m', \text{pred}\ m \to \\ \qquad\qquad \text{unchanged\_on foot}\ m\ m' \to \\ \qquad\qquad \text{pred}\ m' \end{array} \right\}$$

notation: $m \vDash P \triangleq P.\text{pred}\ m$

conjonction

$$P * Q \triangleq \left\{ \begin{array}{l} \text{pred} = \lambda\, m.\ (m \vDash P) \wedge (m \vDash Q) \\ \qquad\qquad \wedge \text{disjoint}\ P.\text{foot}\ Q.\text{foot} \\ \text{foot} = \lambda\, b\, ofs.\ P.\text{foot}\ b\ ofs \vee Q.\text{foot}\ b\ ofs \end{array} \right\}$$

Separation logic in CompCert

predicate

$$massert \triangleq \left\{ \begin{array}{l} \text{pred} : memory \to \mathbb{P} \\ \text{foot} : block \to int \to \mathbb{P} \\ \text{invar} : \forall m\, m', \text{pred}\ m \to \\ \qquad\qquad \text{unchanged\_on foot}\ m\ m' \to \\ \qquad\qquad \text{pred}\ m' \end{array} \right\}$$

notation: $m \vDash P \triangleq P.\text{pred}\ m$

conjonction

$$P * Q \triangleq \left\{ \begin{array}{l} \text{pred} = \lambda\, m.\ (m \vDash P) \wedge (m \vDash Q) \\ \qquad\qquad \wedge \text{disjoint}\ P.\text{foot}\ Q.\text{foot} \\ \text{foot} = \lambda\, b\, ofs.\ P.\text{foot}\ b\ ofs \vee Q.\text{foot}\ b\ ofs \end{array} \right\}$$

pure formula

$m \vDash \text{pure}\,(P) * Q \leftrightarrow P \wedge m \vDash Q$

States correspondence

Obc: $(me, ve),\ f \in c \in p$
Clight: (e, le, m)

$$\text{match\_states} \triangleq$$

States correspondence

Obc: $(me, ve),\ f \in c \in p$
Clight: (e, le, m)

$$\begin{aligned}
&\text{match_states} \triangleq \\
&\qquad \text{pure}\,(le(\text{self}) = (b_s,\ ofs)) \\
&\qquad * \text{pure}\,(le(\text{out}) = (b_o,\ 0)) \\
&\qquad * \text{pure}\,(\text{ge}(\text{f_c}) = co_{out})
\end{aligned}$$

self pointer

out pointer

output structure

States correspondence

Obc: (me, ve), $f \in c \in p$
Clight: (e, le, m)

$$
\begin{aligned}
&\text{match\_states} \triangleq \\
&\qquad \text{pure}\,(le(\text{self}) = (b_s,\ ofs)) \\
&\qquad * \text{pure}\,(le(\text{out}) = (b_o,\ 0)) \\
&\qquad * \text{pure}\,(\text{ge}(\text{f\_c}) = co_{out}) \\
&\qquad * \text{pure}\,(\text{wt\_env}\ ve\ (\text{all\_vars\_of}\ f)) \\
&\qquad * \text{pure}\,(\text{wt\_mem}\ me\ p\ c)
\end{aligned}
$$

the Obc state is well-typed wrt. the context

States correspondence

Obc: (me, ve), $f \in c \in p$
Clight: (e, le, m)

match_states $\triangleq$

$$\begin{aligned}
&\text{pure}\,(le(\text{self}) = (b_s,\ ofs)) \\
&* \text{pure}\,(le(\text{out}) = (b_o,\ 0)) \\
&* \text{pure}\,(\text{ge}(\text{f_c}) = co_{out}) \\
&* \text{pure}\,(\text{wt_env}\ ve\ (\text{all_vars_of}\ f)) \\
&* \text{pure}\,(\text{wt_mem}\ me\ p\ c) \\
&* \text{staterep}\ p\ c\ me\ b_s\ ofs
\end{aligned}$$

memory $me \approx$
structure pointed by *self*

States correspondence

Obc: $(me, ve),\ f \in c \in p$
Clight: (e, le, m)

$$
\begin{aligned}
&\text{match\_states} \triangleq \\
&\qquad \text{pure}\,(le(\text{self}) = (b_s,\ ofs)) \\
&\qquad * \text{pure}\,(le(\text{out}) = (b_o,\ 0)) \\
&\qquad * \text{pure}\,(\text{ge}(\text{f\_c}) = co_{out}) \\
&\qquad * \text{pure}\,(\text{wt\_env}\ ve\ (\text{all\_vars\_of}\ f)) \\
&\qquad * \text{pure}\,(\text{wt\_mem}\ me\ p\ c) \\
&\qquad * \text{staterep}\ p\ c\ me\ b_s\ ofs \\
&\qquad * \text{blockrep}\ ve\ co_{out}\ b_o
\end{aligned}
$$

output of $f \approx$
co_{out} pointed by out

States correspondence

Obc: (me, ve), $f \in c \in p$
Clight: (e, le, m)

match_states $\triangleq$

$$
\begin{aligned}
&\text{pure}\,(le(\text{self}) = (b_s,\ ofs)) \\
&* \text{pure}\,(le(\text{out}) = (b_o,\ 0)) \\
&* \text{pure}\,(\text{ge}(\text{f\_c}) = co_{out}) \\
&* \text{pure}\,(\text{wt\_env}\ ve\ (\text{all\_vars\_of}\ f)) \\
&* \text{pure}\,(\text{wt\_mem}\ me\ p\ c) \\
&* \text{staterep}\ p\ c\ me\ b_s\ ofs \\
&* \text{blockrep}\ ve\ co_{out}\ b_o \\
&* \text{varsrep}\ f\ ve\ le
\end{aligned}
$$

parameters and local variables $\approx$ temporaries

States correspondence

Obc: (me, ve), $f \in c \in p$
Clight: (e, le, m)

match_states $\triangleq$

$$
\begin{aligned}
&\text{pure}\,(le(\text{self}) = (b_s,\ ofs)) \\
&* \text{pure}\,(le(\text{out}) = (b_o,\ 0)) \\
&* \text{pure}\,(\text{ge}(\text{f\_c}) = co_{out}) \\
&* \text{pure}\,(\text{wt\_env}\ ve\ (\text{all\_vars\_of}\ f)) \\
&* \text{pure}\,(\text{wt\_mem}\ me\ p\ c) \\
&* \text{staterep}\ p\ c\ me\ b_s\ ofs \\
&* \text{blockrep}\ ve\ co_{out}\ b_o \\
&* \text{varsrep}\ f\ ve\ le \\
&* \text{subrep\_range}\ e
\end{aligned}
$$

subcalls output
structures allocation